WEATHER DATA INTEGRATION AND ASSIMILATION SYSTEM

Gihan Chanuka Karunarathne

178004U

Degree of Master of Science

Department of Computer Science and Engineering

University of Moratuwa

Sri Lanka

January 2021

WEATHER DATA INTEGRATION AND ASSIMILATION SYSTEM

Herath Mudiyanselage Gihan Chanuka Karunarathne

178004U

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

Department of Computer Science and Engineering

University of Moratuwa
Sri Lanka

January 2021

Declaration

I declare that this is my own work and this thesis does not incorporate without acknowledgement any material previously submitted for a Degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Also, I hereby grant to University of Moratuwa the non-exclusive right to reproduce and distribute my thesis/dissertation, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works (such as articles or books).

Signature: *UOM Verified Signature* Date: 02/01/2021

The above candidate has carried out research for the Masters thesis under our supervision.

Name of the supervisor: Dr. HMN Dilum Bandara

Signature of the supervisor: *UOM Verified Signature* Date: 03/01/2021

Abstract

Numerical Weather Models (NWMs) utilize data collected via diverse sources such as automated weather stations, radars, air balloons, and satellite images. Before using such multimodal data in a NWM, it is necessary to transcode data into a format ingested by the NWM. Moreover, the data integration system's response time needs to be relatively low to forecast and monitor time-sensitive weather events like hurricanes, storms, and flash floods that require rapid and frequent execution of NWMs. The resulting weather data also need to be accessed by many researchers and third-party applications such as logistic and agricultural insurance firms. Existing weather data integration systems are based on monolithic or client-server architectures; hence, unable to benefit from novel computational models such as cloud computing and containerized applications. Moreover, most of these softwares are proprietary or closed-source, making it difficult to customize them for an island like Sri Lanka with different weather seasons. Therefore, in this research, we propose Weather Data Integration and Assimilation System (WDIAS) that utilizes microservices to achieve scalability, high availability, and low-cost operation based on cloud computing. The use of stateless microservices also enables WDIAS to add new features on the fly with rollover capabilities. Moreover, WDIAS provides a modular framework to integrate data from different sources, export into different formats, and add new functionality by adding extension modules. We demonstrate the utility of WDIAS using a cloud-based experimental setup and weather-related synthetic workloads.

Keywords: Cloud computing, data assimilation, data integration, microservice, weather

Dedication

I dedicate this thesis work to teachers, lectures, my family and specially colleagues at Center for Urban Water, Sri Lanka (CUrW-SL). A special feeling of gratitude to my loving parents, Nandawathi Dissanayake and H.M.K. Karunarathne whose put countless sweats to support me throughout my entire life. My wife Jayani Kumarasinghe and my son Sasmitha Karunarathne who missed a lots of wonderful moments to give me freedom to work on this research.

Nevertheless I could not forget all of my friends at CUrW-SL who have supported and be with me during this period of time. Also, special thanks to Dr. Dilum Bandara and Prof. Srikantha Herath for giving me this wonderful opportunity to explore this new domain.

Acknowledgements

I would like to thank the members of my evaluation committee who were more than generous with their expertise and their precious time. Special thanks to Dr Dilum Bandara, my research supervisor for helping me with his precious time, reading, encouragement and been patient with pushing me to the next level. Also, I am, thankful to Dr. Dilika Peris, and Dr. Indika Perera for serve as my evaluation panel without any hesitation and Dr. Srikantha Herath for giving more knowledge on weather domain and support as my external supervisor.

I would like to thank the Department of Computer Science and Engineering at the University of Moratuwa for allowing me to conduct my research and provide all the assistance requested. Special thanks go to the academic and non-academic staff of the department for their continued support. I also thank the Center for Urban Water, Sri Lanka (CUrW-SL) for providing the domain expertise, access to resources, and financial support during the research.

Finally, I would like to thank the lectures, the evaluation panel and the colleagues who helped me with this project. Their enthusiasm and willingness to provide feedback support me to go far beyond my limits and complete my research with an enjoyable experience.

Contents

Declaration i

Abstract ii

Dedication iii

Acknowledgements iii

List of Figures vii

List of Tables ix

List of Abbreviations x

1 Introduction 1

1.1 Motivation 2

1.2 Problem Statement 3

1.3 Objectives 3

1.4 Outline 4

2 Literature Review 5

2.1 Delft-FEWS flow forecasting system 5

2.2 Linked Environments for Atmospheric Discovery 12

2.3 Data Integration and Analysis System 17

2.4 Meteorological Assimilation Data Ingest System 18

2.5 Meta Scientific Modeling 20

2.6 Summary 22

3 Research Methodology **24**

3.1 High-level Design 24

3.2 Architectural Decisions 27

3.3 Microservice Architecture 32

3.3.1 Smart Endpoints and Dumb Pipes Microservice Pattern 33

3.3.2 Database per Microservice Pattern 33

3.3.3 Distributed Transactions over Microservices Pattern 34

3.3.4 The Scale Cube Concept 34

3.3.5 WDIAS Microservices 35

3.3.6 WDIAS Application Programming Interface 38

3.4 WDIAS Database Structure 39

3.4.1 Key Attributes of Timeseries 40

3.4.2 Timeseries Metadata Storage 43

3.4.3 In-memory Database 44

3.4.4 Timeseries Database 45

3.4.5 Network Common Data Form 45

3.4.6 Document-oriented Database 46

3.5 Data Prepossessing 46

3.5.1 Interpolation 47

3.5.2 Transformation 49

3.5.3 Validation 49

3.5.4 Extension API for Data Prepossessing 50

3.5.5 Extension Scheduler 53

3.6 Query Timeseries 53

3.7 Summary 54

4 Performance Analysis **56**

4.1 Test Plan 56

4.1.1 List of Test Plans 59

4.1.2 Performance Metrics 59
4.2 Workload Generation 60
4.2.1 Experimental Setup 63
4.2.2 Configure Experimental Setup on Cloud 68
4.2.3 Performance Tuning 69
4.3 Performance Test Observations 71
4.3.1 Load Testing with Hourly Resolution Data 71
4.3.2 Load Testing with 30-minute Resolution Data 73
4.3.3 Load Testing with 15-minute Resolution Data 75
4.3.4 Load Testing with 15-minute Resolution Data with Auto Pod Scaling 80
4.3.5 Query Module Load Test 86
4.4 WDIAS Performance Evaluation 88

5 Summary **91**
5.1 Conclusions 91
5.2 Research Limitations 94
5.3 Future Work 96

References **97**

Appendix A WDIAS REST API **103**

Appendix B WDIAS Query API **105**

List of Figures

2.1 Schematic structure of a flood forecasting system including Delft-FEWS and communication among other operational systems 7
2.2 Architecture of Delft-FEWS 9
2.3 Delft-FEWS integration with external models 10
2.4 Layered architecture of LEAD 13
2.5 LEAD system framework 14
2.6 LEAD's service-oriented architecture 15
2.7 DIASs common base application platform. 18
2.8 MADIS data flow 19
2.9 The top-level design and overall architecture of MSM system 22

3.1 Modules of a weather data system. 26
3.2 Kubernetes (K8s) architecture. 29
3.3 WDIAS architecture for handling requests on demand. 36
3.4 WDIAS architecture for handling requests asynchronously. 37
3.5 Separation of WDIAS microservices. 38
3.6 WDIAS database structure. 44
3.7 A generic mathematical function model for weather data preprocessing. 47
3.8 Serial interpolation. 48
3.9 Spatial interpolation. 49
3.10 Time interval aggregation. 50
3.11 Timeseries data validation. 50

4.1 WDIAS load testing plan with changing the request size and RPS. . . 58

4.2 Experimental setup with Apache JMeter. 64
4.3 WDIAS load testing throughput shaping timer RPS configurations. . . 65
4.4 Amazon Elastic Kubernetes Service (Amazon EKS) node setup. . . . 70
4.5 Response time vs threads while load testing with hourly data. 72
4.6 Latency against server hits while load testing with hourly data. 73
4.7 Response time vs threads while load testing with 30-minute of data. . 75
4.8 Latency against server hits while load testing with 30 minute of data. . 76
4.9 Response time vs active threads while load testing with 15-minute of data. 77
4.10 Latency against server hits while load testing with 15-minute of data. . 78
4.11 Transaction throughput vs thread while load testing with 15-minute of data. 79
4.12 Latency against server hits while load testing with 15-minute of data with enabled auto-scaling. 81
4.13 Transaction throughput vs threads while load testing with 15-minute of data with enabled auto-scaling. 82
4.14 Load testing with auto-scaling resource usage of import and export modules over time. 83
4.15 Load testing with auto-scaling number of pods of import and export modules over time. 84
4.16 Load testing with auto-scaling resource usage of database adapters over time. 85
4.17 Response latency over time while load testing query test over 5 minutes. 87
4.18 Response latency times vs threads load testing query test over 5 minutes. 88

List of Tables

3.1 Comparison of features among existing weather data assimilation and integration systems and WDIAS. 31

4.1 Amazon EKS nodes 68

4.2 Throughput and latency of load test with 60-minute data 71

4.3 Throughput and latency of load test with 30-minute data 74

4.4 Throughput and latency of load test with 15-minute data 76

4.5 Throughput and latency of load test with 15-minute data while enabled K8s auto-scaling 80

4.6 Throughput and latency of query test cases with 15-minute data . . . 86

List of Abbreviations

Amazon EKS	Amazon Elastic Kubernetes Service
API	Application Programming Interface
CSV	Comma-separated Values
CUrW-SL	Center for Urban Water, Sri Lanka
Delft-FEWS	Deltares FEWS
DIAS	Data Integration and Analysis System
ESB	Enterprise Service Bus
GRIB	General Regularly-distributed Information in Binary form
JSON	JavaScript Object Notation
K8s	Kubernetes
LEAD	Linked Environments for Atmospheric Discovery
MADIS	Meteorological Assimilation Data Ingest System
Microservice	Microservice Architecture
MSM	Meta Scientific Modeling
netCDF	Network Common Data Form
NWM	Numerical Weather Model
RDBMS	Relational Database Management System
REST	Representational State Transfer
RPS	Requests Per Second
SOA	Service Oriented Architecture
WDIAS	Weather Data Integration and Assimilation System
WRF	Weather Research and Forecast

Chapter 1

Introduction

Weather forecast is essential to reduce the impact caused by natural disasters and to effectively manage natural resources like water. The weather forecasts are performed using Numerical Weather Models (NWMs) that use a set of configuration parameters and the current status of the environment such as temperature, rainfall, and wind direction as the inputs. For example, the Weather Research and Forecast (WRF) [1] model is used to predict the rainfall, temperature, and wind direction. Typically multiple NWMs are executed on the same data with different configuration parameters. Then using the real-time observations, the most fitting model is selected. After removing the bias, the data extracted from a NWM are then fed into another set of models/simulators. For example, FLO2D, a hydrologic model that predicts surface water level, could use precipitation estimates from WRF to forecast flood levels.

Center for Urban Water, Sri Lanka (CUrW-SL) [2] is tasked with the mission to reduce water-related natural disasters in Sri Lanka. However, as the existing weather data integration systems are not open source and not flexible enough to change without much help from the maintainers, the CUrW-SL [3] is implementing own system with the idea of long-term maintainability. CUrW-SL collects real-time data from weather stations and other relevant entities and generates daily forecasts. Then the generated results are shared with relevant public departments responsible for disaster planning, risks mitigation, and disaster management. One of the future goals of CUrW-SL is to open up data to the public enabling both societal and commercial use cases.

1.1 Motivation

To enhance the accuracy of weather forecast, it is necessary to provide reliable and detailed weather data as inputs to NWM. These NWMs utilize multi-modal weather data collected via diverse sources such as automated weather stations, radars, air balloons, and satellite images. Before feeding such diverse data (collected from different sources that belong to different stakeholders) into respective NWMs, it is necessary to convert them to a data format that can be ingest by the models. Moreover, the data integration system's response time needs to be relatively low to forecast and monitor time-sensitive weather events like hurricanes, storms, and floods which require rapid and frequent execution of NWMs.

Weather data processed by a NWM maybe used by several other NWMs, as well as by many third-party applications and researcher. For example, logistic companies could use the processed data with their data models to plan and schedule their deliveries. Agricultural insurance companies can warn the farmers, as well as calculate premiums based on the anticipated weather patterns. Besides, farmers could rely on weather forecasts to plan their work.

Data Integration and Analysis System (DIAS) [4] and Meteorological Assimilation Data Ingest System (MADIS) [5] are two of the well-known Weather Data Integration and Assimilation (WDIA) systems. However, those are proprietary systems. Further, while Deltares FEWS (Delft-FEWS) [6] is free to use, but it is not open source. While a few other data integration systems exists, most of them are also proprietary or closed source. Thus, it is difficult and costly to customize such systems for an island like Sri Lanka which experience different kinds of weather seasons over the year [7]. For example, Delft-FEWS does not support FLO2D model integration, which is used by CUrW-SL to forecast water level. To address such issues, there is also an initiative to implement an open-data open-model framework called Meta Scientific Modeling (MSM). But the existing WDIA systems and emerging systems are also based on the monolithic or client service architecture; hence, cannot benefit from technologies such as cloud computing to achieve high scalability and availability. Moreover, such systems cannot be updated or reconfigured without taking the system off line. Further-

more, these systems are platform specific. For example, while Delft-FEWS can run on either Windows or Linux, downtime is needed to introduce any changes to the system.

Given the above reasons, there is a necessity to develop a WDIA system that could handle large volumes of multimodel data efficiently while providing scalability and high throughput. Moreover, it is desirable to support controlled data sharing with other NWMs and third-party access to enable multiple use cases. Furthermore, such a system should be architectured to benefit from technologies such as cloud computing, microservices, and loosely coupled containerized applications.

1.2 Problem Statement

Handling spatio-temporal weather data is challenging, as it involves large data volumes requiring high computing, storage, and network resources. Moreover, the data integration system needs to be capable of interacting with different data formats including spatio-temporal bulk stream data. Furthermore, it should support geographical and time-based queries, as well as provide third-party access to data with easy integration. Scalability and high availability are also essential to support different use cases and mission-critical nature of applications. Further, users should be able to easily deploy and manage the system while benefiting from technological advancements. In this context, the research problem can be stated as follows:

How to design an extendable weather data integration, assimilation, and dissemination system that is capable of handling large volumes of multi-model weather data while providing high throughput, scalability, and availability?

1.3 Objectives

This research addresses the above problem statement by achieving the following research objectives:

- To develop a platform to integrate weather data from different sources in different formats

- To design and develop a schema to store multidimensional weather timeseries data while optimizing access time, availability, and storage
- To optimize data schema to support geospatial and time-based queries
- To develop an API for third-party data sharing
- To efficiently utilize cloud computing infrastructure with low-cost

1.4 Outline

The organization of the rest of the thesis is as follows; In Chapter 2, the literature review is presented. It discusses the existing weather data integration systems such as LEAD, Delft-FEWS, MADIS, DIAS, and MSM. Solution approach including the proposed system architectures and adaptation with problem occurred are presented in Chapter 3. It provides details on how the WDIAS is designed by following a microservice architecture and its data pre-processing capabilities. Performance analysis is presented in Chapter 4. Here we discuss the experimental setup of the WDIAS and analyze the observations concerning performance metrics. Finally, Chapter 5, summarizes the research work and further discusses the research limitations and future work.

Chapter 2

Literature Review

Many weather forecasting, assimilation, and dissemination systems are developed to reduce the damage-causing by natural disasters such as floods, storms, hurricanes, and even droughts. Other countries are also using those systems while adopting them to specific weather conditions. This chapter presents a literature review on existing systems and their architectural designs. In Section 2.1, we present Deltares FEWS (Delft-FEWS), an open model integration framework. Linked Environments for Atmospheric Discovery (LEAD), a distributed open data handling platform as closed-source software, is presented in Section 2.2. In Section 2.3, we discuss Data Integration and Analysis System (DIAS), which is a common data platform that can manage weather data. Meteorological Assimilation Data Ingest System (MADIS) is another widely used data integration system, which also provides data access with quality controls is presented in Section 2.4. Under each system, we explore their system architecture, scalability, and flexibility.

2.1 Delft-FEWS flow forecasting system

Deltares developed Delft-FEWS [6] in Netherland for operational forecasting agencies. The Delft-FEWS codebase is not fully open source at the moment. The main objective of a forecasting system is to run multiple NWMs, hydrologic models, and hydrodynamic models which are required for the predictions. This approach is known

as a model-centric approach [8]. In the model-centric approach, the inputs need to be in the format specific to the model. Also, the outputs produced by the models are specific to the model; hence, difficult to integrate with other models. For example, FLO2D creates human-readable text files as output. However, the Delft-FEWS uses a more modular approach, which is easier to integrate new models and update the system for new requirements. Thus, Delft-FEWS is an integration framework or middleware for the models, rather specific model workflow for weather forecasting.

A forecasting flow creates a sequence of modeling steps by combining data transformation algorithms. In a forecasting flow, each step processes and feeds data into the next step. The Delft-FEWS is flexible enough to integrate new models and algorithms into the core codebase that can be used to create new workflows [6]. So, the Delft-FEWS framework is capable of integrating models into the system and creating workflows for forecasting.

Haggett [9] summarized the key elements of a forecasting system as detection, forecasting, dissemination, and warning, and responding to the events. The Delft-FEWS focuses on the forecasting step out above four steps. The main objective of this step is to provide additional lead time by forecasting future hydrometeorological conditions [8]. It is a valid argument that providing accurate predictions with more considerable lead time can reduce the level of destruction. Further, the forecasting system should be capable of integrating real-time data from weather data observation networks and disseminate the forecasted results through relevant parties to the warning process.

Figure 2.1 shows the connection between a forecasting system to real-time data integration systems and dissemination systems. The Delft-FEWS can integrate data from different sources via import modules into the central servers, and allow forecasters to forecast and disseminate the results into the dissemination systems. One of the widely practiced use cases is to use meteorological forecast data to estimate precipitation, and then use a hydrological and hydraulic model chain to predict the surface water level affected. The hydrological and hydrodynamic models should design based on geographical surface data. The ground surface should be examined geographically

and divided into catchments. Based on the affected catchment, it can be further divided into sub-catchments to reduce the complexity of the simulation task, and reduce simulation time with parallel processing. Ideally, the forecasting system flexible enough to allow modification of models and data while keeping the system interaction method constant as working with multiple forecasters.

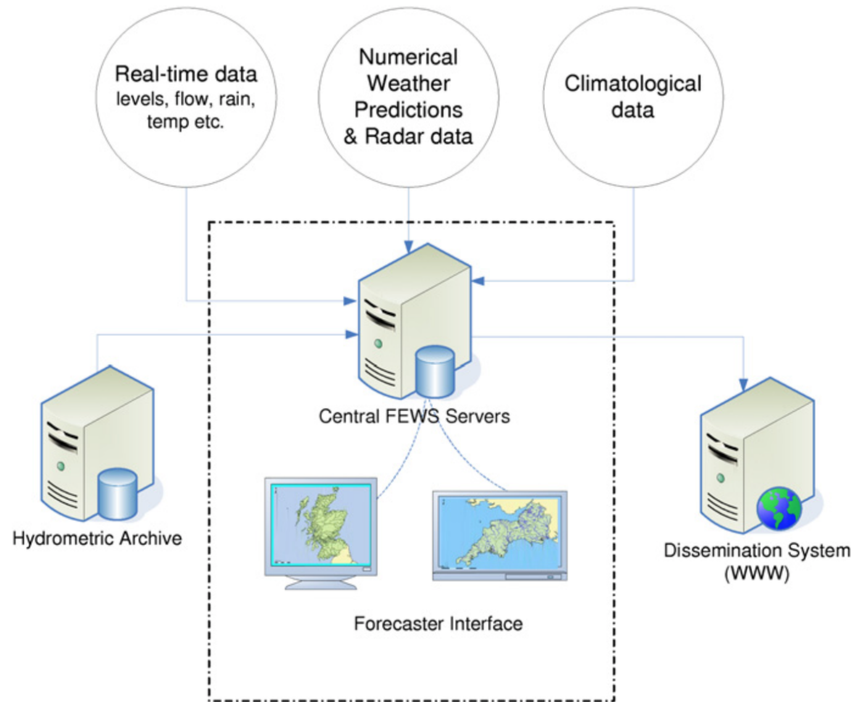

Figure 2.1: Schematic structure of a flood forecasting system including Delft-FEWS and communication with other operational systems [6].

The Delft-FEWS follows a data-centric approach, where it provides a common data-model to interact with other components. All the timeseries data stored in a database by converting them to the common data model. New models are integrated into the system by using one of the interfaces provided to interact with the common data-model [6]. The common data-model allows storing data efficiently without duplicates. Then, the system supports operations like creating forecast reports and sharing the data by integrating new codebase as adapters. However, this approach cannot handle multiple data formats. The Delft-FEWS overcome this issue by introducing adapters for most of the commonly used data formats such as CSV, NetCDF, and JSON. While this enables users to import multiple data formats into the system; it

also introduces additional complexity to the system.

The timeseries data is one of scalar, vector, or grid data type, and different types of data stored as binary objects in a timeseries table inside the Delft-FEWS system. Functional components or any integrated models do not have direct access to the timeseries table, and those should use the data access module to access the data [6]. As explained in the system interpretation, all data access requests need to go through the data access module cause to increase the cost of data access. Because the Delft-FEWS stores different data types as binary objects, there is a penalty for converting data into binary objects and vice versa. Since all timeseries data is stored within a single data source, it allows the system to handle data without much system complexity. However, a single data source becomes a bottleneck when it is required to fulfill all the timeseries data requirements. Further, the performance goes down due to the need to stream spatio-temporal weather data.

Given the above drawback on storing the data in the system, the Delft-FEWS data model uses a set of primary attributes from weather data to identify a timeseries uniquely. Those primary attributes are location, data type, and source of the data [6]. The primary attributes are always required, and there are some secondary attributes to add timeseries metadata further. Also, primary attributes used to index data for making data queries faster. Moreover, Delft-FEWS is able to create separate indexes for each attribute while storing large amounts of data. As an example, the system can separate timeseries data into multiple databases based on the source of timeseries, such as an external source of data or a hydrological model, which creates the timeseries and then index each database on other attributes. Alternatively, separate timeseries by data type and stored in multiple storages give the capability for the system to scale with a factor of identical types. An example of separation of database by data types of scalar, vector, and gridded into three independently hosted databases will increase the performance factor by three times.

Data preprocessing and manipulation are necessary processes in weather forecasting. The data integrated from external sources are not in the appropriate temporal and spatial format that can directly feed as an input to forecasting models or use in other

applications. Therefore, the forecasting environment provides generic data processing steps to convert data into model compatible formats. Such examples are serial and spatial interpolation, data validation, aggregation and disaggregation, and data fusion [6]. This is a vital feature in a forecasting system and affects the quality and accuracy of the predicted data outputs. Because of the common data model concept in the Delft-FEWS, data processing via these functions are much useful. The system contains a set of default data processing functions, but required algorithms can implement as a new Java class integrated with the Application Programming Interface (API) provided by the Delft-FEWS. The Delft-FEWS has language dependency while additional features integration since users need to implement the new functional extensions using Java programming language.

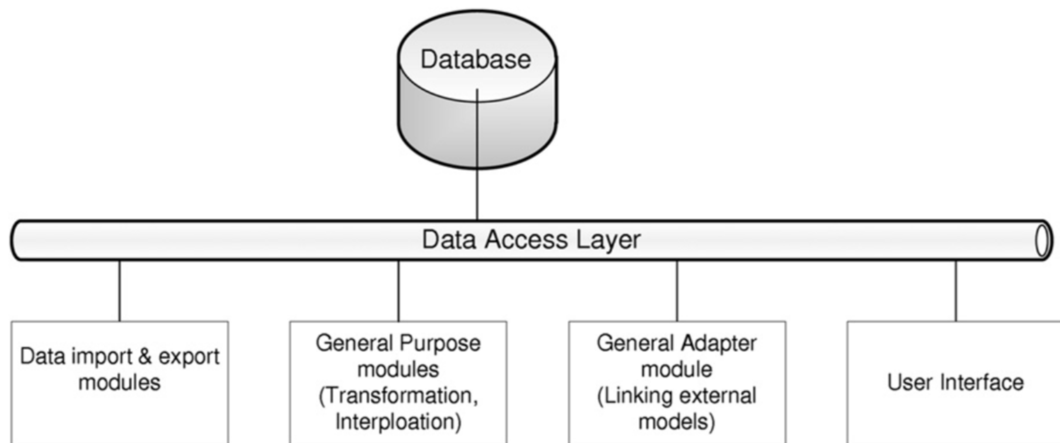

Figure 2.2: Architecture of Delft-FEWS [6].

Delft-FEWS provides data processing and modeling libraries to access scalar and grid timeseries using the same method. However, the database access only allows through the data access layer, as shown in Figure 2.2 [6]. The system depends on a single database instance, and all the requests are coming to the database through the data access layer. It is possible to setup and connect to an enterprise-level database with clustering and shading as a paid solution to enhance the throughput. Still, the design itself inherently suffers from a single database bottleneck when accessing a large set of timeseries by the models.

One of the simple and most useful features in the Delft-FEWS is the use of an open

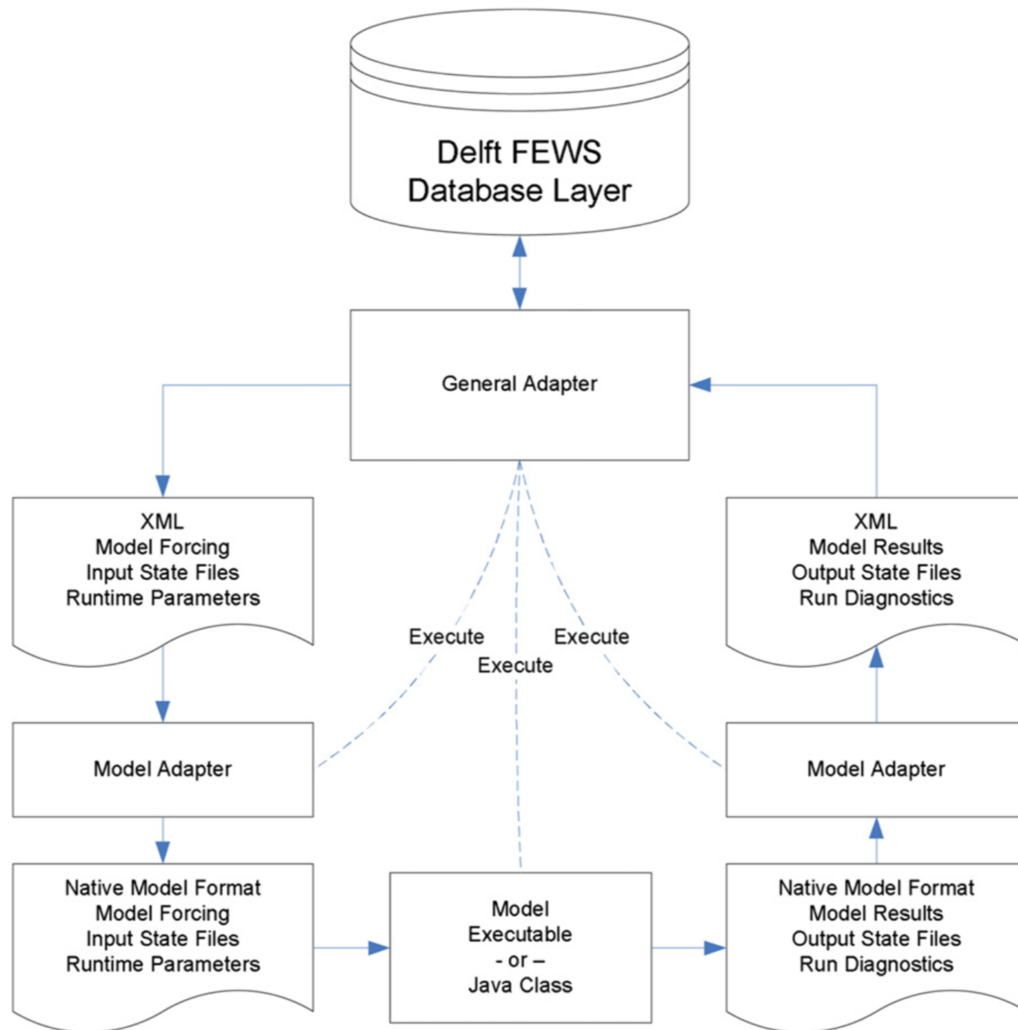

Figure 2.3: Integrating Delft-FEWS with external models [6].

approach to integrate models and data. The concept of the open modeling framework is used by the Delft-FEWS, which is proposed by open model integration [10]. It merely gives the flexibility to allow operators to integrate more models, as well as variations of the same model and come up with new forecasting flows as much as possible.

The users need to configure the input timeseries data via XML configurations, and Delft-FEWS generates the model input data as XML files. As shown in Figure 2.3, a model adapter transforms the data into the required format for the model as a preprocessing step. The Delft-FEWS has prebuilt a set of model adapters, and users can use available model adapters. Next, the Delft-FEWS executes the external model, which generates the model output, and allows users to run multiple models parallel or run

those sequentially as needed. In the post-processing step, the results of the model transform into an XML file, which imports into the database through the data access layer, as shown in Figure 2.3 [6]. The general adapter used by the Delft-FEWS for the model execution is causing tide coupling into the execution process in the system. Users do not have the flexibility to run the models with different configurations, such as parallel execution of a particular model. Those issues maybe overcome by triggering an external process at the execution time. But, it introduces more difficulty in handling the next step in the forecasting flow, after the successful execution of the model.

In this research, we do not focus on implementing workflow management for forecasting, and the user has to come up with their flow mechanism or use an existing scientific flow management system. However, it is a concept that needs to be discussed and understood properly to design a data integration and assimilation system. However, the proposed architecture within the research should be capable of integrating workflow mechanisms with its extendable architecture.

The Delft-FEWS exchanges data with the models using XML files. Thus, it causes XML files to become very large when trying to access a large set of data, which can lead to I/O bottlenecks and causing performance problems [6]. The authors of the Delft-FEWS [6] seem to have noticed and accepted this problem. To overcome the issue mentioned above, they have introduced a file-based exchange of data. The proposed solution includes the use of binary XML files, the streaming of files via memory, and the use of netCDF files. Nevertheless, as far as for the users, it seems to be adding more complexity and context to get higher performance via the system.

If the user is unable to find a model adapter among available Delft-FEWS adapters, then the user has to implement the adapter for the model. The user has the flexibility to create a new adapter within Delft-FEWS, but the effort required to develop an adapter for a model will vary depending on the complexity of the data formats required by the model [6]. If the users want to use an existing adapter and need to configure something not supported via the existing adapter, it is difficult without technical support since source code is not available. Also, there are some cases it is hard to develop an adapter or running alongside the Delft-FEWS. For example, the FLO2D model used

for hydrologic modeling only runs on the Windows platform. If the user sets up the Delft-FEWS on a Linux platform and uses Linux based model adapters, it is hard for users to develop a solution for integrating the FLO2D model.

2.2 Linked Environments for Atmospheric Discovery

Linked Environments for Atmospheric Discovery (LEAD) [11] addresses the fundamental research challenges needed to create an integrated and scalable framework for adaptive analysis and prediction of the atmosphere. The weather forecasting researchers require many computer resources to predict and analyze weather models in the mesoscale with less time. Rather than each researcher run and handle their computer resources for the weather experiments, LEAD provides a shared pool of resources. Then the researchers can use the resource pool to run their experiment in shorter amounts of time and a more massive scale. At the time, some researchers are developing their experiment forecasting flows, they are not using the resources, and other researchers can run their forecasting flows using available resources. The foundation of the LEAD system is to create a dynamic workflow orchestration and data management in a web services framework [11].

LEAD's complex range of services, applications, interfaces, and IT resources, local and external networks, and storage is compiled by users in workflows to study mesoscale incremental manner [11]. As it follows the SOA, all the features are implemented as independent services. The above implementation enables LEAD to scale up each service as required and update each service without affecting other services. New models that are required for implementing a new forecast flow also have to be implemented as a service and then integrate into the system.

Figure 2.4 shows the key capabilities of the LEAD system. The top layer of the LEAD enables users to search and acquire information, simulate and predict weather conditions using digital atmospheric models, assimilate, analyze, use and visualize data and output of models.

The second layer contains fundamental tools such as the following to help to link

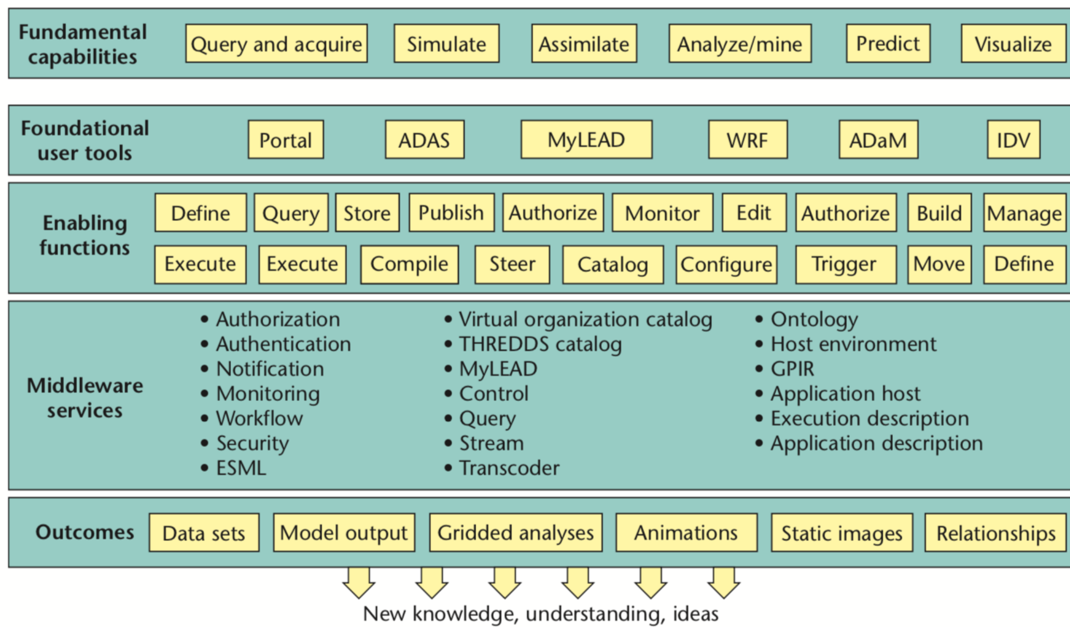

Figure 2.4: Layered architecture of LEAD [11].

services together [11]:

- The users interact with LEAD via a web portal
- Data quality control and assimilation happens via ARPS Data Assimilation System (ADAS)
- myLEAD is the metadata catalog
- Weather Research and Forecast (WRF) [1]
- Algorithm development and mining is a set of tools for analyzing observational data, assimilated data sets, and model output
- Integrated data viewer

The LEAD follows a concept of the workflow orchestration for on-demand, real-time, and dynamically adaptive systems called WOORDS. The system is doing the workflow orchestration as given in each procedure. The forecast researchers define the procedural rules and feed them into the system. The system is trying to act immediately after the submission, and also it transmits the data, runs the models, and sends back

the put with a lower time delay as possible. Based on the system load and priority of the workflow, the system will schedule the workflows automatically.

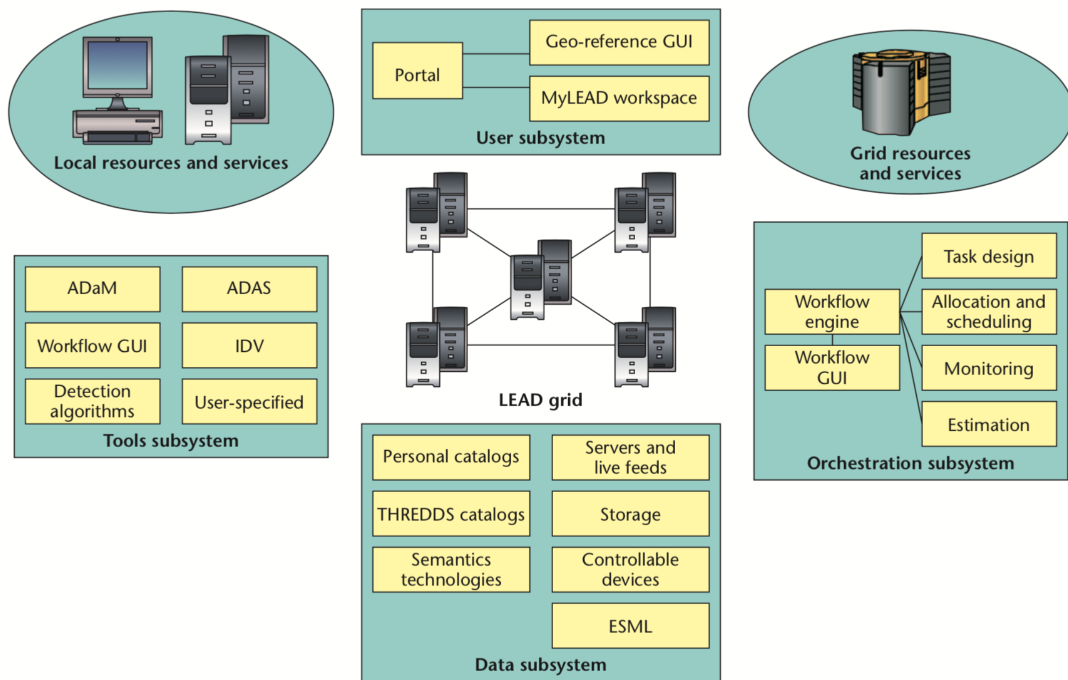

Figure 2.5: LEAD system framework [11].

As shown in Figure 2.5, LEAD consists of the following sub-components, as well as provides a distributed system for integrating and testing LEAD's components:

- User subsystem comprises the LEAD portal and enable user can access services
- Data subsystem manages data and metadata, all output of digital models produced by operational or experimental models, and information generated by users
- Tools subsystem provides all IT tools and meteorological tools
- Orchestration subsystem offers the technologies with which users can manage data flows and model execution flows and can create and own output

As shown in Figure 2.6, LEAD SOA has five different but highly interconnected layers. The bottom layer represents the calculation, application, and raw data sources that are distributed in the LEAD grid and elsewhere. The next level supports web

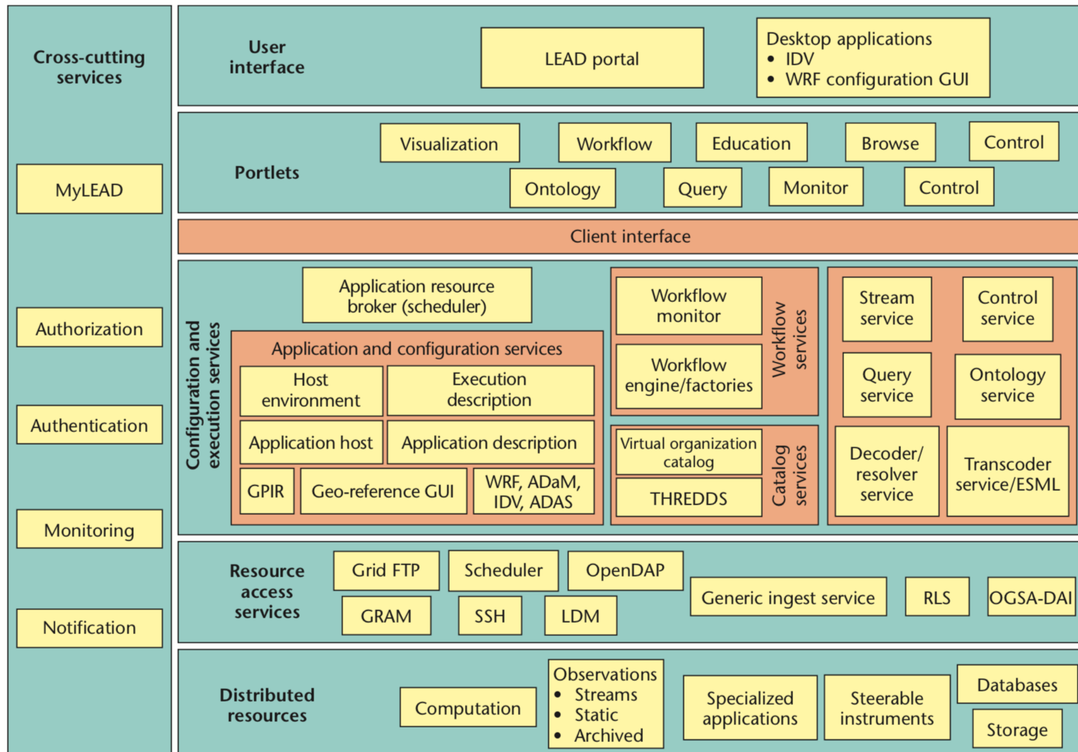

Figure 2.6: LEAD's service-oriented architecture [11].

services that provide access to raw services [11]. These two layers are working together since LEAD system resources are distributed over multiple locations and creating a pool of resources. The upper layer of the raw resources is abstracting the complexity of managing and accessing the resources and provides simplified access. In the upper layers, the underline system is viewed as an unlimited resource pool for storing and handling data.

The configuration and execution services of the middle layer, consisting of five elements, represent the services that call LEAD workflows. These are some critical aspects of a weather data management system. Most of the services listed below are required for creating workflows for weather data forecasting. While analyzing the LEAD, we realized that it is essential to understand the following basic needs of a weather data management system.

- The application-oriented configuration service manages the implementation and execution of real applications, such as the WRF simulation model, ADAS, and ADaM tools [5]. While creating weather forecast workflows, the researchers

require to change the behavior of the model by changing some of the configurations of the model or run a different version of the model.

- The application resource broker, which matches the appropriate host for execution to each application task, based on the execution time constraints [5]. This service is a critical part of the LEAD system and responsible for using the resource pool in an optimized manner. One important fact is, while we are designing weather data systems, it is required to increase the capacity of the system automatically, and manage the system.

- The workflow engine service, which manages the experimental workflow instances, invokes both the configuration service and the application resource broker [5]. The above service is a part of workflow orchestration.

- Catalog services represent how a user or an application service discovers data products in the public domain or LEAD services [5]. Most systems need a catalog service and essential to have such features because users need to search for the availability of the data, and users want to analyze existing data for decision making.

- Users need a multitude of data services to support comprehensive query, access, and transformation operations on data products. An essential goal behind LEAD is the transparency of access that facilitates user requests on all available heterogeneous data sources without the negative effects of different formats and naming schemes [5]. Users need to have transformation services to read data in different formats. The query service gives the capability to search via available data without any effect on the data formats.

The top of Figure 2.6 shows the user interface of the LEAD, and each user gets individual access to services via the interface. After the user login into the system, the LEAD system allows portlets to access the services on half of the users based on the authentication and authorization setting of the account.

LEAD is a more advanced system which can support mesoscale weather forecast with the effort of multiple universities in the United State America with the effort

of many researchers and using many computer resources. At the time of building, it uses the SOA architecture into its depth, and at the implementation, each service can scale as needed and possible to enhance the service without interrupting other services. However, during our research, we only focus on creating a framework that is extendable as a system needs more features similar to LEAD services. When compared to the Delft-FEWS, the LEAD is more dynamic and auto scaling system which is capable of using a distributed large resource pool. But it is a proprietary system, thus others cannot get benefit of it, and the user need to own the resource pool and manage them. In critical situations the system need more resource for the forecasts, and users need to pre-owned the resources to get ready for such scenarios. But with modern cloud computing concepts, users can lease much as resources to handle the critical situation, and release the resource after that while only paying for the utilized resource duration.

2.3 Data Integration and Analysis System

Data Integration and Analysis System (DIAS) [4] collects and archives data from satellites, weather stations, numerical forecasting models, and climate projection models. Once these data integrate into useful geographic information, DIAS generates results for managing global environmental problems and natural disasters [4]. The DIAS can assimilate weather data from multiple data sources with different data formats. After successfully storing the data, it generates results which can be used for different kinds of purposes by other parties.

The data storing mechanism is similar to the other systems discussed above. The DIAS provides an API to import the data and store them in a large array of storage after converting to the required data format. The APIs work to convert or reformat data into manageable forms and are useful for creating the DIAS storage archive. The DIAS's API contains various tools, including the real-time data exploration and archiving system, the data download system, the quality control system, and the metadata management system [4]. To store on the large volume of disk space, it converts the data via a

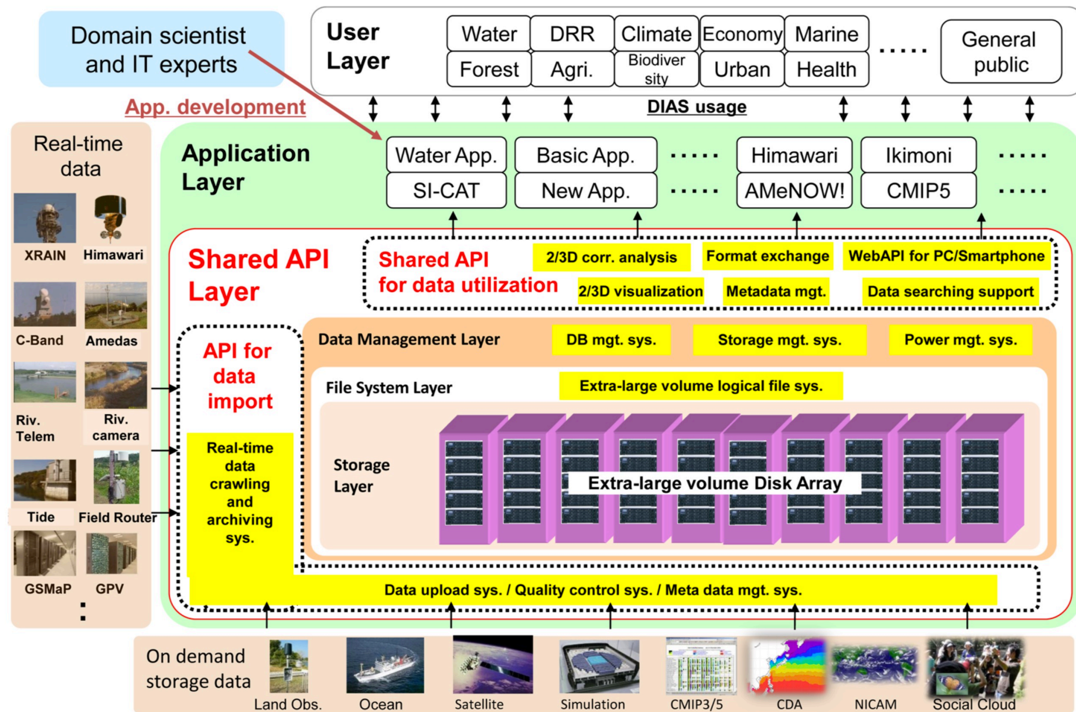

Figure 2.7: DIASs common base application platform [4].

provided set of APIs. The above concept is similar to Delft-FEWS's open data model approach, but it has extra-large storage volume and a database management system which abstract for the upper layer to access the data. Further, the DIAS contains additional tools that provide the functionality of quality control of the data and metadata management, which is similar to some of the services in LEAD.

As shown in Figure 2.7, it offers an application layer that allows scientists to work on research tasks and to develop specific tools and applications by collaborating with IT experts [4]. Once the data is stored on the DIAS system, it shares those data via the shared API layer and allows users to remote access to the data and easy integration via the APIs.

2.4 Meteorological Assimilation Data Ingest System

Meteorological Assimilation Data Ingest System (MADIS) [5] is a data integration and assimilation system as the name implies that collects data from dozens of suppliers,

then checks the quality of the data and saves it in netCDF format. Later, users can access the data with the support of the different types of data access protocols.

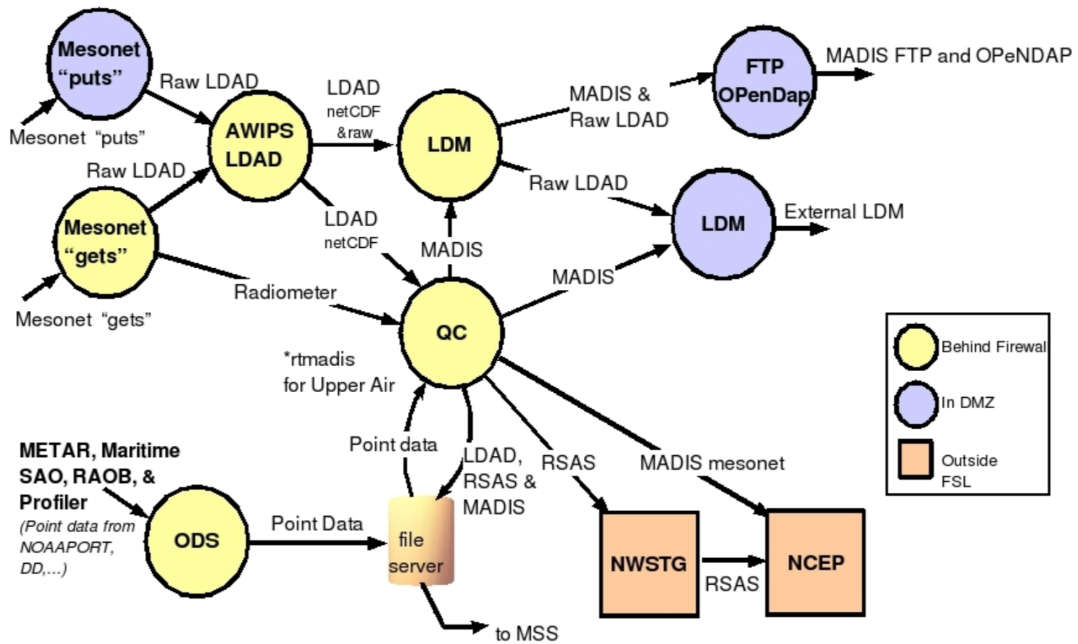

Figure 2.8: MADIS data flow [5].

MADIS contains a distributed architecture for data acquisition, processing, and distribution functions. Besides, the MADIS was developed based on the distributed architectural patterns available at the time of implementation. The various functional hosts have been paired using Linux High Availability (HA) to provide automated failover in the event of a system failure. It uses several methods for data dissemination such as FTP, Local Data Manager (LDM), and the Web OPeNDAP (OPen source project for Network Data Access Protocol) [5]. It uses functional calls like Remote Procedure Call (RPC) to invoke among the cluster of nodes. In Figure 2.8, the data is transported from the input and preprocessing systems to the central computer systems and then to other hosts for storage and distribution [5]. To get high availability, it has added some updates to its architecture, as mentioned in the paper.

Based on the above statements, we can conclude that these kinds of functionalities are available at the modern cloud computing tools out of the box, and those tools are providing the scalability and high availability. Using those tools, it is possible to implement such a system with less effort and high confidence.

The MADIS regularly acquires mesonet data from dozens of network providers representing more than 14,000 stations, and the system translates into more than 30,000 station reports every hour. Different systems that work inside and outside the firewall collect the data. The collected data is sent to the data server of the Advanced Weather-Interactive Processing System (AWIPS) of the central installation for processing and conversion to a common data format, NetCDF. The NetCDF files are then transferred to MADIS compute nodes [5]. After reading the statistics of the MADIS system, I realized the paper is a useful reference for system architecture design and its performance testing. When compared to the Delft-FEWS, the MADIS also converts the assimilated data into netCDF format. After converting to netCDF format, the netCDF files are transferred to the compute nodes and accessible via OPeNDAP. Many of these computers are configured using Linux clustering with high availability [5].

Some of the MADIS data is considered to be the property of the supplier, and data access should be provided based on the authorization. To address the issue, MADIS data is divided into six different versions, depending on the access level authorized by the data provider [5]. When compared to Delft-FEWS, MADIS provides data assimilation with control-based access to the data.

2.5 Meta Scientific Modeling

Meta Scientific Modeling (MSM) framework developed an open system architecture that consists of open-data concept to access heterogeneous data sources via data agents, and open-model concept to allow management of individual model via model agents. Even though other systems like Delft-FEWS provide these features, they do not provide decentralized model execution and data integration. Specifically, this framework provides a graphical user interface for workflow creation similar to LEAD, rather than providing a configuration-based workflow system as in Delft-FEWS. MSM tries to effectively address the drawbacks of the existing modeling systems by providing features such as an independent open-source and graphical-based workflow engine, facilitating users to obtain data directly from popular sources, decentralized model administration

by modelers, and enables models to be freely connected with each other in the MSM framework. However, in our research, we focus on data ingress and egress aspects than the workflow engine.

Figure 2.9 shows the MSM framework from a top-level design point of view, and it is mainly composed of the MSM core which is an interface with the workflow engine, data agents, and model agents. Thus, the MSM core can lead to a bottleneck and a single point of failure in critical situations. The MSM core also controls all other components, and by using corresponding model agents and data agents allows it to connect with external models and remote data sources. The MSM core is connected to the workflow engine and provides the end-user with all workflow control capabilities, and it contains data persistence services that can be used by all MSM components. Another drawback is, implementing new data agents and model agents are programming language dependent. To implement model and data agents, users need to inherit MSM classes in the provided language. Then the users need to upload the code into a folder (not a containerized mechanism, or continuous service deployments).

The open-data architecture of MSM is used to access multiple and heterogeneous external data sources. The users are able to integrate data from any source via data agents, but the data agents are depending on inheriting a generic class to do so. Which means there is a language dependency. The open model architecture allows users to develop and deploy their own model via model agents. Here users also have to use a generic agent provided by the MSM framework, but the models can be implemented independently. Users can deploy new model agents by copy-pasting the source code to a specific folder, but it is possible to run the models written in languages such as C, C++, FORTRAN, Python, and Java. The model execution follows a flow similar to the Delft-FEWS model adapter with a prepared input file, execute the model, and processing output files. The MSM framework has a data management layer that stores the data as an in memory and local data caching, featuring a non-relational database scheme that supports high throughput. It stores particular timeseries data as msmDataSets, and creates a unique identifier for each msmDataSet. The msmDataSets storage framework contains the time-steps. Each time-step contains a reference to the msmDataSet

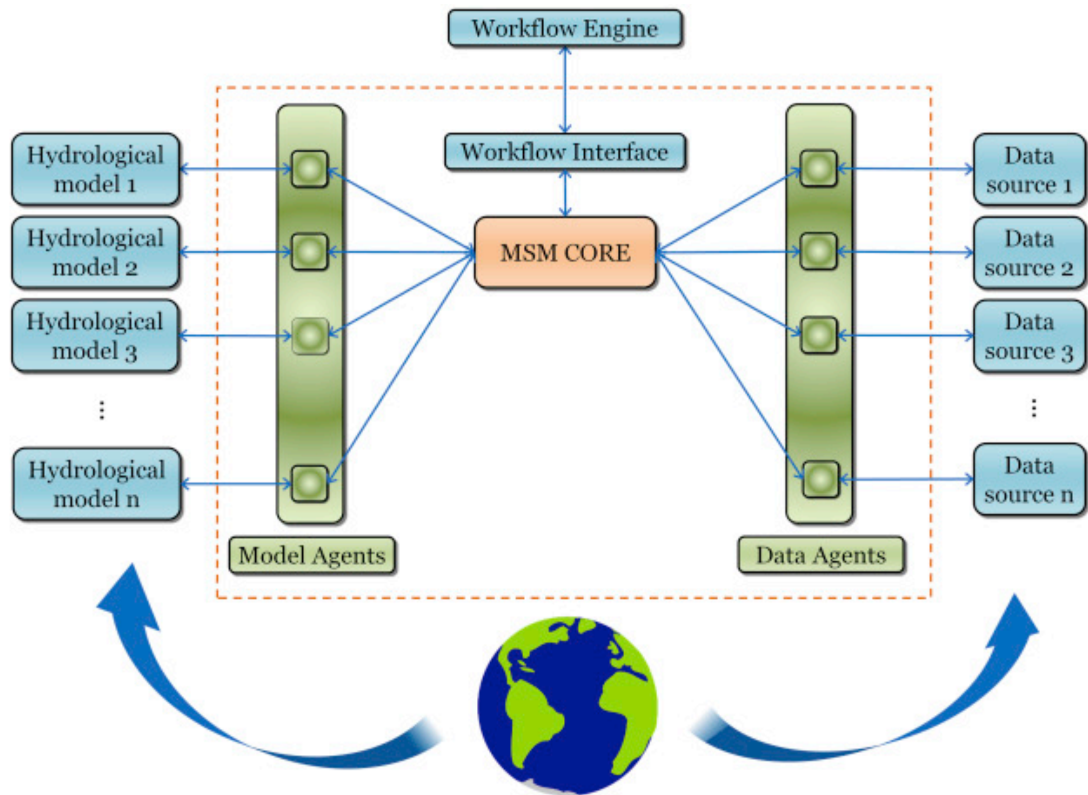

Figure 2.9: The top-level design and overall architecture of MSM system [12].

it belongs to, and a document (the data-oriented structure) with a 2D image.

While the authors presented a use case of using the MSM system, detailed evaluation regarding its performance was not given. Nevertheless, it demonstrates the usefulness and effectiveness of the framework in various aspects of scientific modeling activities. The MSM has only been fully tested on Windows 10 operating system, which means the framework has some operating system dependency. Further, details on how it parallellize data agents, model agents, and the MSM core are not presented. Even though the authors mentioned that MSM follows a decentralized agents approach, it seems to have a monolithic distributed architecture based on the available data.

2.6 Summary

We presented several related works on WDIA system implementations such as Delft-FEWS, LEAD, DIAS, MADIS, and MSM. During our research, one of the significant

issues that we focused on, store the data ingested from different sources with different formats efficiently. The Delft-FEWS and MADIS are using the netCDF as the common data format to store the data, which is one of the best solutions available with modern technologies as well. Even the LEAD and DIAS papers are not mentioned directly; those systems are also following a method of having common storage space to store bulk data. The Delft-FEWS, LEAD, and MSM have the capability of handling forecasting workflow, but other systems are mainly focused on providing a weather integration and data management system. The Delft-FEWS provides a modular approach via its general adapter, and according to the SOA architecture of LEAD provides the same functionality via services.

Above WDIA systems are developed with the contributions from lots of researchers, and were implemented specifically to address issues like natural disasters. A use case such as CUrW-SL, a given WDIA system requires to remodel the systems and define the forecasting flows to support the country-specific requirements. Even some of the systems try to follow a generic approach to integrate new modules, but it is challenging to deploy modules for a closed source or proprietary software. Further, most of the systems depend on the underline software platform and do not support more resource-efficient and highly-scalable technologies such as cloud computing.

Chapter 3

Research Methodology

This Chapter presents the design decisions taken while implementing Weather Data Integration and Assimilation System (WDIAS) and how we adopt from the systems analyzed in Chapter 2. Section 3.1 presents a brief discussion on how the weather forecasting is happening and why does it need to interact with multiple data formats. Section 3.2 shows how the architectural design of the proposed solution evolved while designing and implementing WDIAS. Section 3.3 presents microservice architectural concepts followed while developing WDIAS. Then the reasons behind developing WDIAS database structure and technologies used are discussed in Section 3.4. Section 3.5 discusses how WDIAS enables data preprocessing and adding capabilities using extensions. Finally, Section 3.6 includes details on how to perform timeseries search and geo-based queries on WDIAS.

3.1 High-level Design

To get an overall idea of the weather forecast flows, let us look into one of the weather forecasting flows in the CUrW-SL [2].

1. First, the researchers fetch the weather forecast data from global model which is in the GRIB format. After extracting GRIB data, they run the WRF with geographical coordinates for a given area, and generates WRF model forecast in netCDF format.

2. Then they extract forecast precipitation weather data in netCDF to CSV file format as scalar timeseries for each interested location. Next they feed forecasted precipitation data to HEC-HMS hydrodynamic model and get the water discharge forecast of a water canal as the output in the CSV file format.

3. They use a developed 2D FLO2D hydrologic model for a given area to simulate the water level. By feeding HEC-HMS's water discharge forecast and WRF precipitation forecast as Grid files into the FLO2D hydrologic model, the researchers forecast the water level of the modeled area as ASCII Grid files after the simulation is completed.

4. By uploading above water level forecasts in ASCII Grid files into a geographic information system after convert to a compatible format, they calculate the loss estimation.

In the above weather forecasting steps, we can see that the weather data need to convert to different formats while feeding from one model or step to another. After the models run successfully, the resulting output data may not be accurate or biased based on different factors. In such situations, we need to correct predicted weather data by comparing them with accurate real-time weather data. We are required to collect real-time data from automated weather stations and accumulated weather data over a given period by automatic water level stations.

Most weather data collecting devices use own data formats and different standard protocols to transmit the data. Then we need to validate the data and modify the data to remove errors. Also, these data need to convert into model compatible data format before feeding into models. The models output data in different formats and need to store those bulk data efficiently, and the system should be flexible enough to upgrade to support new data formats easily. Essentially, the system should share the real data, forecasted data, and other created data with relevant parties based on authorization.

Figure 3.1 shows the basic functions of a weather data integration and assimilation system consisting of the following modules:

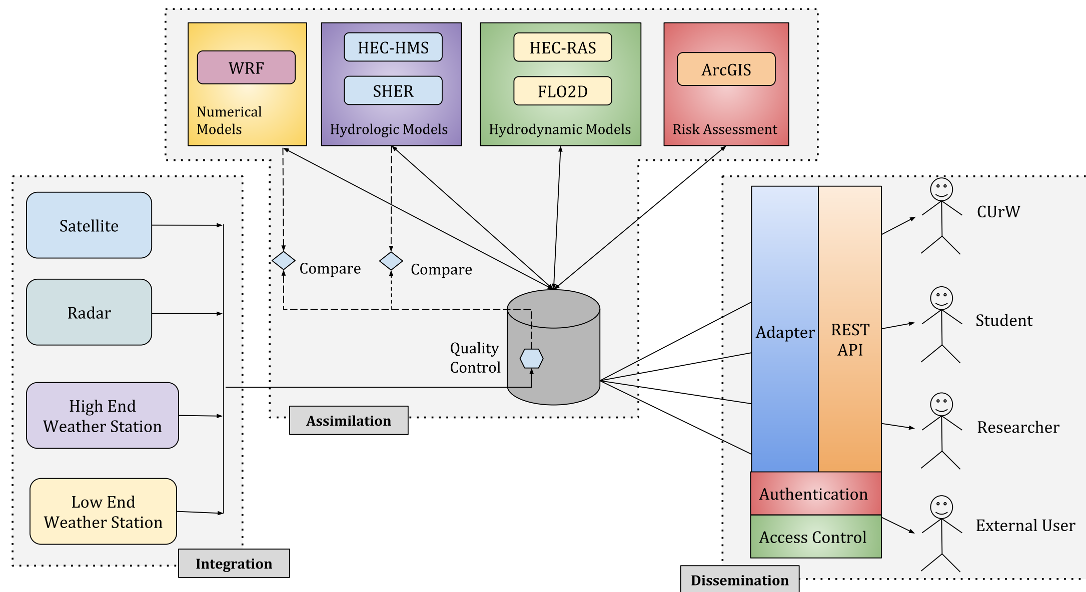

Figure 3.1: Modules of a weather data system.

Integration

The system should be capable of integrating data from different sources such as satellite data, high end, and low-end weather station. Also, the system should be able to handle multidimensional spatial and temporal weather data efficiently and optimally.

Assimilation

Then the system should be able to fulfill weather models varying data requirements. Also, those models reproduce a large set of redundant data. Thus, the system should store the bulk data while optimizing the disk space.

Dissemination

Then different users should be able to retrieve data as they require. Also, the users should be able to easily search into the available information in the system based on timeseries metadata or based on geo-queries.

3.2 Architectural Decisions

The initial design of WDIAS used the Service Oriented Architecture (SOA). Moreover, it tried to use an Enterprise Service Bus (ESB) to integrate different modules, as ESB could act as a shared layer for all of the modules to interact together. By default, ESB also provides publisher/subscriber capabilities.

The ESB processing units called *mediators*. A mediator can take a message, carry out some predefined actions on it, and output the modified message. These mediator capabilities can be used to integrate the modules of a weather data system, as shown in Figure 3.1. ESB also supports different transportation protocols such as HTTP and web sockets. However, we realized that ESB is not suitable for data streaming or bulk data processing. Also, ESB suffers from a single point of failure, as all the messages are going through a common shared bus and slower communication (cannot be used for transfer data). In the second design phase of WDIAS, we attempted to use the actor model using the AKKA framework [13] to overcome above ESB drawbacks. At the same time, we moved away from the SOA architecture design to the microservice architecture design.

Thus, we redesigned the system architecture with decomposing into microservices and used actors to implement each microservice in the system. In that case, we mapped each microservice in WDIAS to an actor or an actor with slave actors. SOA focuses on imperative programming, while the microservices architecture focuses on a reactive actor programming style using faster messaging mechanisms. When compared to SOA, the microservice architecture has several advantages:

- Follow the principle of a single responsibility.
- Resilient/flexible because a failure of service does not influence other services. If you have monolithic or significant service errors in a service/module, this can have an impact on other modules/functionalities.
- High scalability because demanding services can be deployed on different servers to improve performance and stay away from other services so that they do not

affect other services. It will be challenging to achieve the same with a single large monolithic service.

- Easy to improve the system because the microservices are decoupled from other services. It is also easier to change and test each microservices.
- Little impact on other services because each service is independent.
- Easy to understand because they represent a small functionality.
- Ease of deployment.

The AKKA framework has some of the disadvantages of implementing each microservice as an actor, as described in AKKA documentation [14]. One of the attractive features of microservice architecture is the independent nature of the microservices. The above concept allows choosing different technologies for each microservice based on the advantage that could bring in. Also, the microservice architecture will enable us to independently develop and maintain a system by multiple smaller and more focused teams/communities. But using the actors as microservice, the actors communicate using message passing cause to result in a too-tight code coupling between the services and difficulty to deploy services independent of each other, which is one of the main reasons for using a microservices architecture [14]. To overcome these problems, we moved to the concept of container orchestration based microservice architecture.

Containerization is an alternative to virtualization that supports virtual machines or hypervisors. It includes encapsulating or grouping the software code and all its dependencies so that it can work uniformly and consistently in any infrastructure [15]. The concept of containerization and process isolation has emerged after the open source Docker Engine [16], an industry standard for package software into standardized units for development, shipment, and deployment.

WDIAS used Kubernetes (K8s) as the container orchestration system. K8s is an open-source tool for automating deployment, scaling, and management of containerized applications [15]. It groups the containers that make up an application into logical units for easy management and detection. Using K8s, we deploy each microservice as a container and manage it by providing scalability and availability.

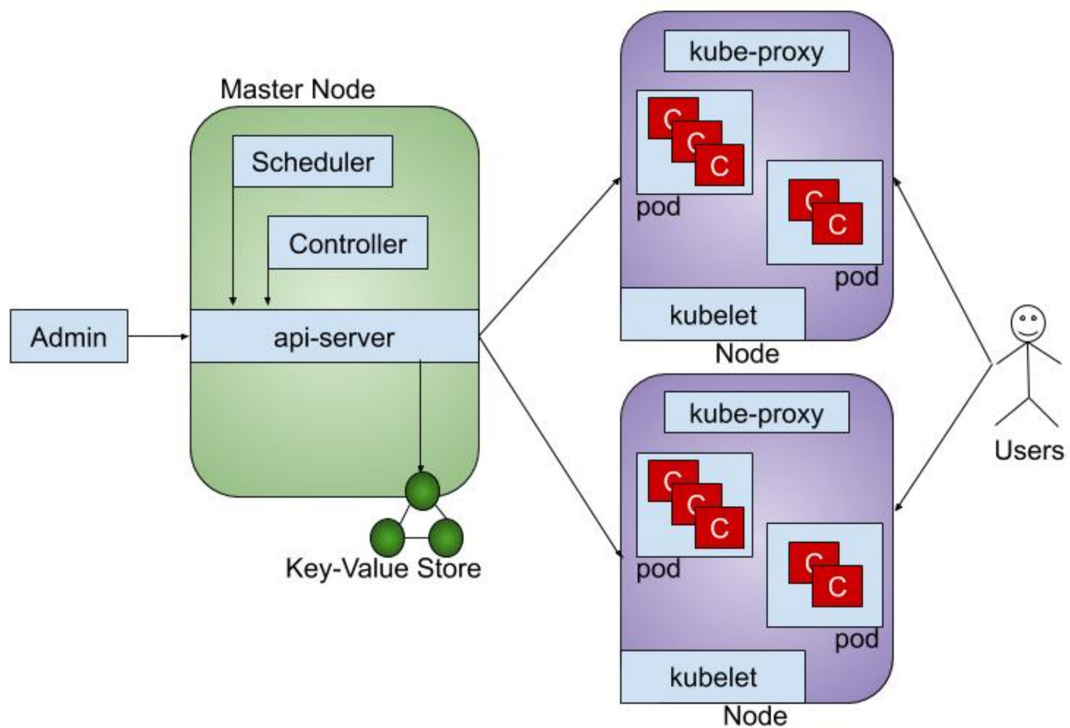

Figure 3.2: Kubernetes (K8s) architecture.

Figure 3.2 give an overall idea on the components of K8s,

- Pods – Cluster of containers that can group other container images in a single unit.
- Nodes – The machines (VMs, physical servers, etc) in a cluster that runs your applications and cloud workflows.
- Kubelet – An agent that runs on each node in the cluster. It makes sure that containers are running in a pod.
- Kube-proxy – A network proxy that runs on each node in the cluster, and maintains network rules on nodes.
- Kubernetes master – Responsible for maintaining the desired state for your cluster.
- etcd – Consistent and highly-available key value store used as K8s backing store for all cluster data.

Users can add nodes much as required into the K8s cluster, and K8s manage and deploy applications as pods into cluster nodes. Each microservice can deploy as a pod, which is a group of containerized applications. K8s is able to scale as much as the user requirements by deploying multiple copies of the same pod. By separating each microservice as a container, and running them as multiple pods inside the K8s removes the tight coupling of microservices when compared to the actors. It preserves the independent nature of the microservices and supports the high scalability of the system.

Table 3.1 shows a comparison between existing systems and WDIAS concerning software architectural aspects. Here, we compare systems under categories such as data storage, performance, architecture, and other features. The Delft-FEWS and MSM frameworks store the data on a central data service, which could become a bottleneck while handling large volumes of data in a critical situation. Other systems can use a pool of resources allowing them to increase the performance by adding more resources to the system. Regarding the performance of the systems, we consider scalability, availability, cloud computing support, and system deployment capabilities. All the systems can scale up to some extent by adding resources beforehand or in real-time. However, LEAD, DIAS, and WDIAS could scale on-demand by adding more resources and support automatic elasticity (i.e., scaling up and down). LEAD, MADIS, DIAS, and WDIAS have high availability, as they use a resource pool that can run on different locations. Further, these systems are capable of automatically handling partial system failures enabling fault tolerance. When we consider cloud computing support, the system should support easy integration with clouds. It is possible to rent virtual servers and deploy the local services in the cloud infrastructure, but some systems may not allow access to them remotely. Even though some systems can deploy in such a manner, they may not be capable of automatically allocating resources on demand. Hence, we cannot consider that those systems can utilize cloud infrastructures fully. Furthermore, we can easily maintain a system if it consists of independent subsystems, and each subsystem is capable of deploying or upgrading independently. Independent service deployment is a more flexible and better way to handle this, rather than copy-

Table 3.1: Comparison of features among existing weather data assimilation and integration systems and WDIAS.

	Functionalities		Desirable	FEWS	LEAD	MADIS	DIAS	MSM	WDIAS
Data Storage	Framework data storage mechanism	Central data storage		X				X	
		Resource pool	X		X	X	X		X
		Flexibility of database structure			X				X
	Independent of underline databases				–				X
Performance	Scalability	Scale by adding more resources	X	X	X	X	X	X	X
		Scaling on demand by adding more resources	X		X		X		X
		Elasticity	X		X				X
	Availability	High availability	X		X	X	X		X
		Fault tolerance	X		X	X	X		X
	Cloud computing	Easy cloud integration/migration	X		–				X
		On-demand resource allocation	X						X
	System deployment	Continuous deployment without downtime	X		X	–	–		X
		Independent subsystem deployments	X		X				X
		Support a proper dependency handling mechanism		X				X	
Architecture	Architecture pattern	Distributed client server		X					
		Distributed monolithic				X	X	X	
		SOA	X		X				
		Microservice	X						X
	Loose coupling	Subsystem independency	X	X	X	–	X	X	X
		Programming language independent subsystems	X		–		–		X
	Extendability	Architectural independency to ingregrate new subsystems			X			X	X
		Extendable data integration support	X	X	X	X	X	X	X
	Availability of source code		X					X	X
	Data preprocessing	Data preprocessing		X	X	X	X	X	X
		Programming language independent preprocessing modules	X		–		–		X
		Add new preprocessing capabilities without system downtime	X		X	–	–	–	X
		Run preprocessing with auto scaling	X		X				X
Other	Workflow engine			X	X			X	
	Model execution and integration			X	X			X	

An ”X” in the table represents a feature offered by the corresponding tool. An ”–” in the table represents that information available are not enough to make an statement, and empty presents a feature is not offered.

pasting codes to deploy changes.

We discuss the architecture of the systems in terms of their design, dependencies of the subsystems, extensibility, availability of source code, and data preprocessing capabilities specific to the weather domain. While other existing systems use distributed client-server architecture or monolithic distributed architecture [17], LEAD uses the SOA architecture. However, WDIAS has the advantage of using modern cloud computing technologies to implement microservices as containerized applications. The architecture implementation of independent subsystems allows users to maintain subsystems independently. Further, WDIAS enables different subsystems to be implemented using different technology stacks. Specially WDIAS and MSM provide an open-source framework that gives users the capability to enhance or modify the system as per the requirement. These systems provide data preprocessing capabilities for weather data and the ability to add new preprocessing capabilities. However, WDIAS is also capable of providing auto-scaling with data preprocessing. Workflow engine and model execution support are some other extra features that are helpful with weather forecasting. However, the WDIAS focuses on implementing a WDIA system with supporting main features, and it provides an extendable open framework to integrate such features easily.

3.3 Microservice Architecture

A microservice architecture style is an approach to developing a single application as a package of small services, each working in their process and communicating using lightweight mechanisms, often through the delivery of a Representational State Transfer (REST) API. These services are built around commercial capacities and can be used independently by fully automated implementation machines [18]. In a microservice architecture, each microservice implements decoupling from other microservices and minimizing the centralized dependencies. Thus, we can use different programming languages and data storage technologies to implement each microservice.

There are lots of microservice patterns developed and evolved. Through the fol-

lowing sections, we will discuss some of the microservice design patterns mostly used in the context of designing the software architecture for our research in brief. Next, we will discuss how microservice architecture promotes only to create endpoints that solely belong to each service logical domain. Next, consider only using a database per service, which allows it to be loosely coupled between systems and achieves high scalability. Then we will discuss how distributed transactions can makeover microservices. Last, we will discuss a concept called *the scale cube*, which explains the scalability from the perspective of x-axis, y-axis, and z-axis.

3.3.1 Smart Endpoints and Dumb Pipes Microservice Pattern

When building communication endpoints, many architectural approaches are adding significant smart by providing different protocol supports and complex logic support through the endpoints. As an example, ESB supports multiple protocols, and endpoints can be used to implement complex logic. But microservice architecture follows an alternative approach called as *smart endpoints and dumb pipes* [19]. Applications built based on microservice architecture are intended to be as disconnected and as coherent as possible. Each microservice has its domain logic and independent of other microservices logic and works more like filters in the classical Unix pipelines. Each microservice provides its functionalities as a REST API instead of using complex protocols. After receiving a request for each microservice through its REST API, it applies the correct logic and produces a response.

3.3.2 Database per Microservice Pattern

Microservice prefers to have each microservice manages its database, either different instances of the same database technology or entirely different database systems - *polyglot persistence* [20].

The idea is to keep the persistent data of each microservice private for this service and only to make it accessible through an API. There are different ways to keep the permanent data of a microservice private. But it is not required to set up a database

instance for each microservice. For example, when you use a relational database, the following methods can be used to keep the data private [21]:

- Individual tables per service – each microservice has a set of tables that restricts access for other microservices.
- Schema per service – each service has a private database schedule for a given microservice.
- Database server per service – each service has its database server.

Private tables per service and *schema per service* have the lowest overhead. The use of a schema per service is interesting because it makes the property clearer. Some high throughput services may require their database server.

3.3.3 Distributed Transactions over Microservices Pattern

Every microservice can have its database to guarantee a loose couple between microservices. Maintaining data consistency between services is a challenge because two phase-commit/distribution transactions are not an option for many applications. To archive distributed transactions, a service publishes an event when the data changes. Other services use this event and update their data [22]. The above approach is also called the *saga pattern*.

3.3.4 The Scale Cube Concept

The scalability of a system can be explained via a concept called *the scale cube*, which talks about the scalability of the application by defining over X, Y, and Z axes.

X-axis scaling – When scaling an application through the x-axis, multiple instances of the application execute behind a load balancer. If there are n copies, then each copy handles $1/n$ of the load. The above simple approach is commonly to scale a service.

Y-axis scaling – When compared to the z-axis and x-axis scaling, which follows the concept of running multiple identical copies of the application, but y-axis scaling segregates the application into multiple services in the same domain. So, the end services are responsible for one or more closely related functions of the previous service before the segregation. There are different ways to split the application into services.

- Verb-based decomposition. e.g., checkout.
- Decompose the application by the noun. e.g., order management.
- It is possible to use a combination of verb-based and noun-based decomposition for an application.

The microservice architecture is an application of y-axis scaling.

Z-axis scaling – The z-axis scaling follows the concept of each service running the same application code (similar to x-axis scaling). The big difference is that each service is responsible for only a subset of the data. z-axis splits are commonly used to scale databases. As an example, data partition (aka., sharded) across a set of servers based on an attribute of each record. Even our research is a design based on the y-axis scaling; it is possible to use the z-axis scaling to further scaling the system. We will discuss more detail in Section 4.4.

3.3.5 WDIAS Microservices

Figures 3.3 and 3.4 show the system interaction of the WDIA microservices architecture while handling requests. Each circle in the figures represents a microservice in WDIAS, and each microservice is implemented as a containerized application. The left side of Figure 3.3 shows the import modules of WDIAS, and the right side shows the export modules. Also, we implemented each module as a microservice to independent maintainability and upgradability. We followed the concept of smart endpoints and dumb pipes explained in Section 3.3.1, and each import microservice only does the specific task of converting and forwarding the request to the correct data adapter module. For scalar, vector, and grid data types, we created adapter microservices that

are optimized for storing such kinds of data. The metadata data of the timeseries store using an RDBMS, which gives more performance over retrieving timeseries metadata. The metadata is cached using an in-memory database to fast access while implementing the database structure. The system generates a unique identifier for each timeseries, and throughout the WDIAS system, all microservices use it to handle data for fast access.

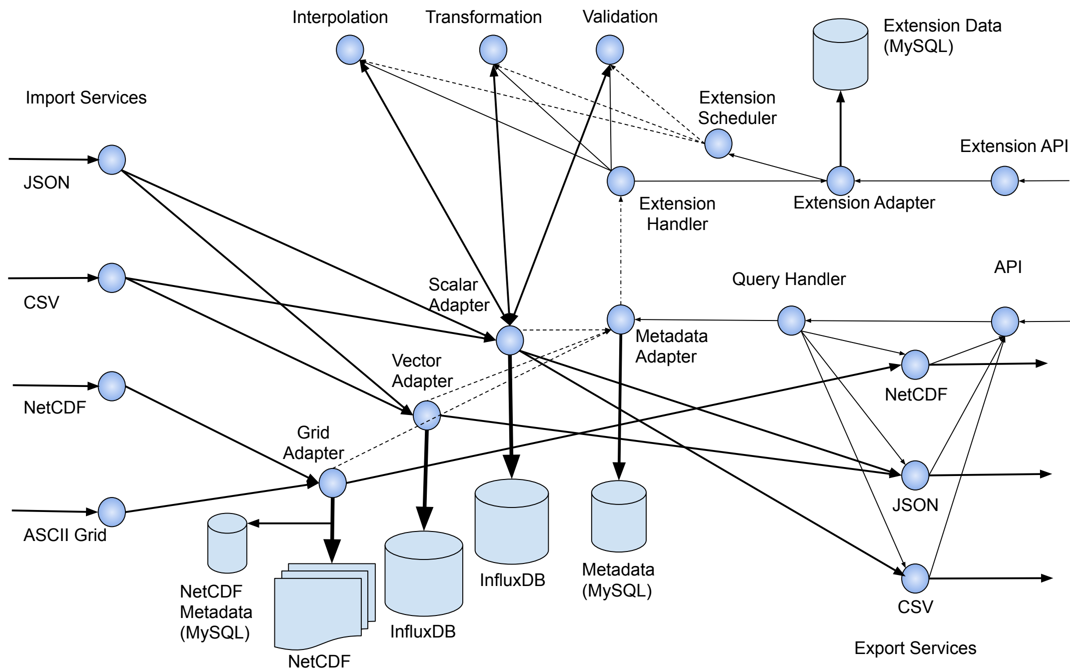

Figure 3.3: WDIAS architecture for handling requests on demand.

Export module microservices also follow the concepts of single responsibility and provide the capability to export the data into required formats of the weather models. Each data type adapter follows the microservice architecture concept of database per service, as mentioned in Section 3.3.2 When the system requires more database throughput, it is easy to scale each database instance by adding more resources or using z-axis scaling concepts.

As shown in Figure 3.3, when a request with a smaller payload comes to the system, the system handles the request on-demand and response back. However, as shown in Figure 3.4, when a request with a larger payload comes to the system, it stores the data for an asynchronous process and responds with a unique id that can be used to verify

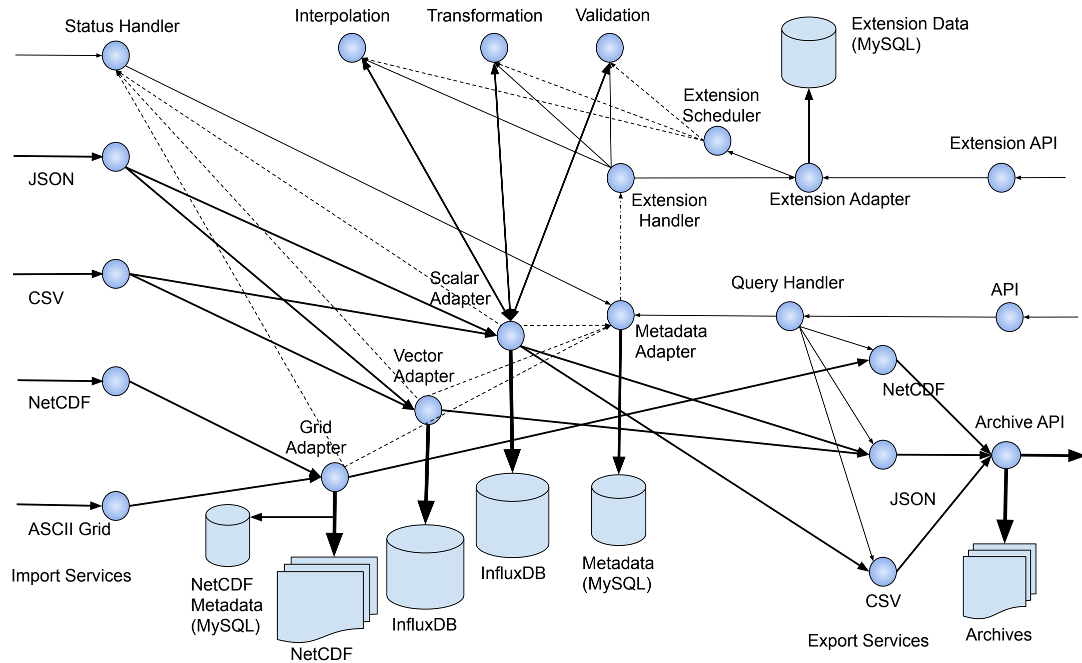

Figure 3.4: WDIAS architecture for handling requests asynchronously.

whether data processed successfully or not. We used the microservice concept of saga pattern, as mentioned in Section 3.3.3, while processing the grid data. First, it stores the data and responds with a unique identifier to the user. After successfully storing the data, the microservice publishes an event. Another microservice listens to those events and processes the data and updates the system status.

Figure 3.5 shows the clear separation of microservices into the modules of WDIAS. Each adapter has a separate database, and the database is hosted separately for the high performance and to avoid interference. Apart from that, as we further discuss in Section 3.5, the extension modules are running independently, such as interpolation, transformation, and validation, and we will also discuss in Section 3.5. The extension adapter allows users to register new event-based or time-based triggers for the extensions. The extension scheduler triggers the events based on time and the extension handler triggers events based on data change.

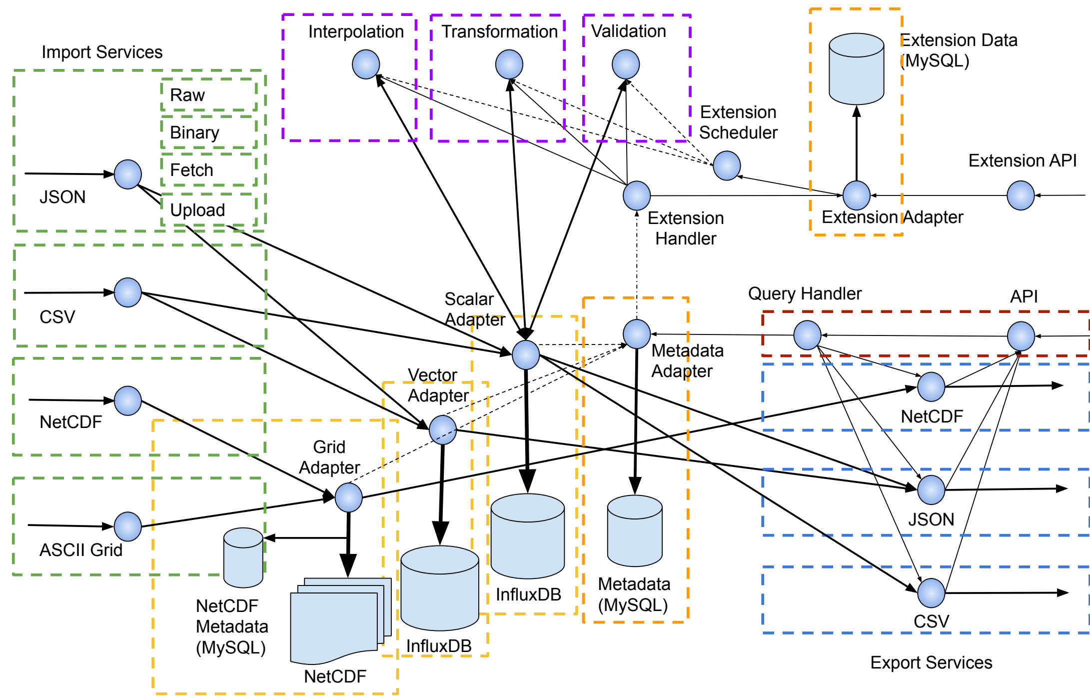

Figure 3.5: Separation of WDIAS microservices.

3.3.6 WDIAS Application Programming Interface

We defined an REST as the communication layer for WDIAS, and it affects the microservice architecture design as well. It follows a simple REST API and allows users to interact with the system through HTTP methods.

We provided a set of timeseries endpoints to register new timeseries metadata before storing the real data, and a detailed description appears in Appendix A. Timeseries endpoints allow users to interact with metadata adapter to create and update the timeseries metadata. After creating timeseries metadata, we cached on the query handler to perform search queries and geo-indexed to support geo-based queries. When a user registers a timeseries metadata, the system generates a unique timeseries identifier based on the metadata values. Then the user can use that identifier to store the real weather data and retrieve stored data.

Current WDIAS only supports handling point locations and grid locations. If it needs to extend the system to store irregular grid data, then it should add another set of location endpoints for storing irregular grid location details.

Other than the endpoints in Appendix A, we exposed another set of endpoints

to configure extensions, and we will describe further in Section 3.5. Also, other module endpoints start with the module name such as import, export, and extension when compared to timeseries metadata endpoints. Using the above convention, we allowed adding a new module system if required. For import and export modules, the names of the endpoints defined as the modules name then follow the data format. The above endpoint naming convention provides the flexibility to integrate new data format modules for import and export. Then the endpoint name ends with the type of data operation. Data can upload as a single file with binary or multiple files with the upload. Other than that, users can send requests with raw data with required headers has mentioned by a particular import module. Thus, the API convention of `/MODULE/DATA_FORMAT/DATA_TYPE` allows users to integrate new import or export modules into the system, as well as add new data types.

3.4 WDIAS Database Structure

The architecture and the storage management of WDIAS is designed and developed based on the weather data. We need to look into the weather data timeseries deeply to understand the underline design decisions of the WDIAS database structure.

Timeseries – *A timeseries is simply a series of data points ordered in time*. In the weather domain, we are interested in observation and forecasting timeseries data. Each timeseries, the data points can be formed in different formats as well. For example, scalar (0D), vector (1D), grid (2D), and polygon (2D).

Based on the above interpretation, a timeseries have data points ordered in time. Other than that, timeseries have additional information that is known as *timeseries metadata*, which can use to identify one timeseries from another uniquely. As an example, we can place a weather station on a *location*, and it can measure different types of weather data that are called *parameters* such as temperature, precipitation, and wind direction. Each parameter can produce different timeseries. Even we can use the location as metadata to identify the timeseries, but that is not enough to distinctly identify the parameters. Thus, we need to use a set of attributes to identify a time-

series uniquely. Next, we describe key attributes that are used by WDIAS to identify a timeseries uniquely.

3.4.1 Key Attributes of Timeseries

Module ID

String field which describe the source of the data generate. e.g., HEC-HMS, FLOW2D, weather-station.

Value Type

Scalar, vector, grid are the value types that are interested in WDIAS.

Location

Location of the timeseries. All locations have a unique String identifier called *locationId*. Further, two types of location types are interested in WDIAS, namely:

- *Point locations* – Contains a name that is human readable. latitude(lat) and longitude(lon) of the location on the earth surface. Following is an example of define a location using JavaScript Object Notation (JSON).

```
1 {
2     "locationId": "wdias-hanwella",
3     "name": "Hanwella",
4     "lat": 6.909722222,
5     "lon": 80.08166667
6 }
```

- *Regular Grid locations* – Contains a name that is human readable. The grid presents by dividing into equal size cells. Thus, it needs the number of rows and columns. And the location of the first cell and the width and height of it. Below JSON format shows an example of the regular grid location.

```
1 {
2     "locationId": "wdias_kelani_basin",
3     "description": "Kelani Basin",
4     "rows": 120,
5     "columns": 139,
6     "geoDatum": "Kandawala",
```

```
 7     "gridFirstCell": {
 8         "firstCellCenter": {
 9             "x": 397074.0,
10             "y": 504875.0
11         },
12         "xCellSize": 250.0,
13         "yCellSize": 250.0
14     }
15 }
```

- *Irregular Grid locations* – (endpoints are available within the WDIAS system, but skipped the implementation since it is out of the interest of the scope.)

Parameter

Parameter describes the variable measuring against a location. All parameters should have a unique string identifier called *parameterId*. A parameter can be used in multiple locations. While defining a parameter, three required fields need to be provided, such as variable, unit, and *parameterType* as shown in the below example JSON format.

```
1 {
2   "parameterId": "O.Precipitation",
3   "variable": "Precipitation",
4   "unit": "mm",
5   "parameterType": "Instantaneous"
6 }
```

- *Variable* – Nature of the variable measuring. Example precipitation, temperature and water level
- *Unit* – metric unit of measuring
- *Parameter Type* – Should be one of instantaneous, accumulative, or mean

Timeseries Type

Users can define timeseries type while setup the system. We determined the following types as the default values. Timeseries type can be a combination of a source followed by category:

- *Sources* – External or simulated. *External* means whether timeseries is taken from an external source, and *simulated* means it generated by simulations of the users.
- *Category* – Historical or forecast. *Historical* means whether timeseries have continuous data such as observations from a weather station. *Forecast* means discrete set of timeseries data produce by forecasting.

Time Step

All time steps has a unique String identifier called timeStepId. The unit should be one of Second, Minute, Hour, Day, Week, Month, Year, or NonEqualDistance. One of multipliers or dividers can be used to define the interval between each measurement. We can represent each minute time step in following JSON format.

```
1 {
2     "timeStepId": "each_min",
3     "unit": "Minute",
4     "multiplier": 1,
5     "divider": 0
6 }
```

Among the above key attributes; location, parameter and time step attributes are composite attributes. But each of them has a unique identifier. Given that, a timeseries can uniquely identify by *moduleId, valueType, parameterId, locationId, timeseriesType and timeStepId*. Here is an example of an unique weather data timeseries using JSON format.

```
1 {
2   "moduleId": "HEC-HMS",
3   "valueType": "Scalar",
4   "parameterId": "O.Precipitation",
5   "locationId": "wdias_hanwella",
6   "timeseriesType": "External_Historical",
7   "timeStepId": "each_hour",
8 }
```

Even if a timeseries consists of data points in time ordered, the data for each data point can vary based on the valueType. The scalar data point consists of a single value. The vector data point consists of two values, such as magnitude and the direction of the vector. But for grid data, a point can consist of multiple values. To store data efficiently,

we used a set of different databases based on the advantage of using them for each value type. Other attributes do not have much impact on classifying the timeseries data.

It is difficult to search for the timeseries data if those are stored over multiple data sources and trying to avoid having duplicates. Thus, it adds low overhead to have a single data source to look into while searching for the timeseries metadata. The efficiency of searching for timeseries metadata can increase by using a caching mechanism. To support search queries with lower latency, we used an index for the fast search for timeseries metadata, and a geo-based index to support geo-based search queries.

Many modern database systems give a higher performance on a specific type of application. Thus, we need to use the correct database system based on application requirements. Considering the aforementioned database requirements, we cannot get all of the benefits using a single general database system available. Thus, we created a database structure after carefully choosing database systems per the system requirements and combining their capabilities. According to the design principle of microservices architecture, we created microservice for each of the value type. Here we followed the concept of *database per service* as described in Section 3.3.2. Section 3.3.5, those microservices are grouped as *adapters* that share an API to interact with while keeping the persistent data private.

3.4.2 Timeseries Metadata Storage

Figure 3.6 shows WDIAS's database structure. The following sections describe the database structure of WDIAS with compare to the design with more details on how is support the microservice architecture.

To reduce the overhead while searching for timeseries and handling duplicates, we used a timeseries metadata store. Since metadata consists of multiple key attributes, the data can be efficiently stored using a Relational Database Management System (RDBMS). It helps to store data without any duplicates and optimize the data with normalization. As an example, the parameter attribute can only have a dozen possible values. Using an RDBMS, those values can be stored once and reuse with defining

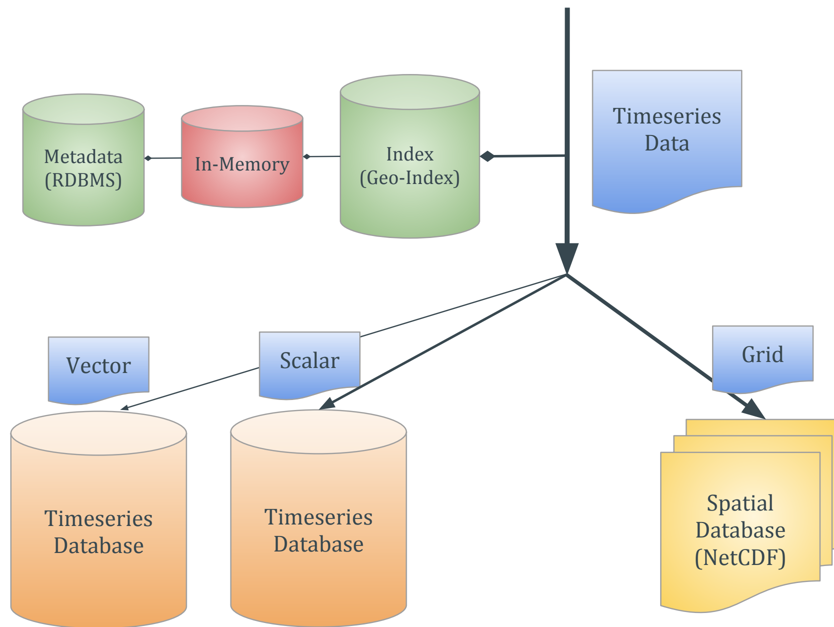

Figure 3.6: WDIAS database structure.

timeseries against location via references.

Since one of WDIAS main goals is to use open source tools, it used MySQL as a persistent database for storing timeseries metadata described above. One RDBMS instance used only for metadata adapter service. Other than that, another RDBMS instance used for extension adapter. We will explain in more detail in Section 3.5. We used MySQL only for the above services, and those database instances keep the consistency of the schema for timeseries metadata and extension metadata. Further, we used an in-memory database instance for caching the requested metadata to reduce the load on the RDBMS database instances while repeated queries. Other than MySQL, many open-source SQL databases can be used to replace the same functionality.

3.4.3 In-memory Database

We used Redis [23] as the in-memory database since it is a widely used database system with excellent support. It also supports Publisher Subscriber capabilities as an inbuilt feature. Redis is an open-source tool, and clustering is also available for scalability. Redis uses in-memory caching for fast access to the frequent access data in metadata adapter and extension adapter. Other than that, status adapter uses Redis as a key-value

pair store to maintain the status for asynchronous requests handling while storing grid data. The extension adapter uses Redis's publisher subscriber capabilities to create a cronjob in extension scheduler service when a new extension creates with a trigger on time during data preprocessing.

3.4.4 Timeseries Database

Among several timeseries open source databases, InfluxDB [24] uses a unique database indexing algorithm to gain higher performance while storing timeseries data when compared to other modern databases available. Also, it has better support, higher usage, and following supports,

- Open timeseries database with MIT license
- Support SQL-Like query language
- Clustering available with a commercial version

Other than InfluxDB, the ElasticSearch database can be used, but it mainly uses indexing for searching. InfluxDB allows us to change the read/write behavior and indexing mechanism via database configurations. Further, users can use the commercial version of InfluxDB to gain more performance, like database sharding and replicas.

Two individual instances of InfluxDB used for scalar adapter and vector adapter services in WDIAS system. For the scalability, the system uses *database per service*, as mentioned in Section 3.3.2. If users want further performance, the database can split into another set of partitions based on other key attributes based on the z-axis scaling described in Section 3.3.4. As an example, scalar and vector adapters can further split into more services based on other key attributes such as timeseries type.

3.4.5 Network Common Data Form

netCDF [25] is a self-describing, machine-independent data format that supports the creation, access, and sharing of array-oriented scientific data. Moreover, NetCDF also supports parallel file access. NetCDF is widely used in the scientific domain due to

the capability of storing multi-dimensions data efficiently and flexibly. Thus, we used NetCDF for storing the grid data due to its capability of storing array-oriented scientific data efficiently when compared to other database systems.

3.4.6 Document-oriented Database

To provide fast geo-search queries, we have to use a database system that supports geo indexing. We used MongoDB, which is a document-oriented database with geo indexing capabilities. Also, MongoDB [26] is a general-purpose, document-based, distributed database, and supports,

- Supports query operations on geospatial data [26]
- Clustering and sharding available for scalability and reliability

We used MongoDB geospatial features to support geo-queries in WDIAS. Also, we indexed the timeseries metadata with other indexing features to serve external time-series metadata queries. If particular timeseries metadata not present in the query adapter, then fetch metadata from the metadata adapter and indexed in the MongoDB. Vise a versa; when a new timeseries created in the metadata adapter, it triggers an event to index metadata in the query adapter. Here we used the *y-axis scaling* concept by splitting metadata search into multiple services. Within the system, it uses in-memory database cache to support internal metadata access, and external search queries fulfill by the query adapter with using MongoDB.

3.5 Data Prepossessing

The data imported via import modules may have incorrect data, missing data, and incompatible data which cannot directly feed into simulation modules. Thus, we need to preprocess imported data before using them for weather forecasting. The WDIAS supports data preprocessing capabilities via extension modules. The current system has a few inbuilt extensions such as interpolation, transformation, and validation, but users can add more extensions as required.

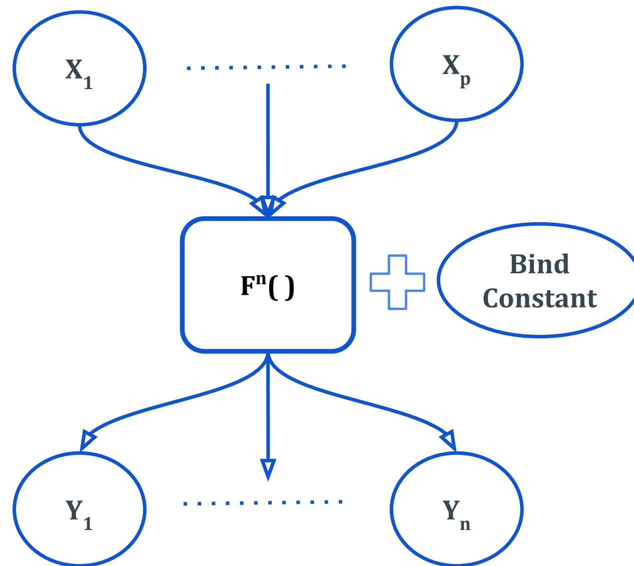

Figure 3.7: A generic mathematical function model for weather data preprocessing.

The data preprocess functionalities support simple generic mathematical function models. Each extension is considered as a mathematical function that can take `p` input timeseries variables and output `n` timeseries variables after applying the given function. Further, the system supports to provide a bind constant as configurations to the extension at the time of preprocessing. Thus, users can change the behavior of an existing function by providing different bind constants at the time of creating new triggers for an extension and produce different behaviors. Next, we explain how to derive interpolation, transformation, and validation functions using the generic mathematical function model described above.

3.5.1 Interpolation

The interpolation extension generates data at desired locations or desired data points using a serial or spatial interpolation technique. It is used to fill gaps in linear measured data and to derive spatially distributed data for weather timeseries, such as precipitation and temperature based on information available at nearby locations. Here, we will discuss two types of interpolation, such as serial interpolation and spatial interpolation, below.

Serial Interpolation

During serial interpolation, we use interpolation to fill any gaps in a linear timeseries. The interpolation module will only consider a single timeseries, which is itself filling the missing data points. For example, as seen in Figure 3.8, there can be timeseries that has data points of temperature collected using a weather station. There can be missing data in this timeseries, and we can use a simple linear interpolation algorithm to fill the missing values based on the existing data points.

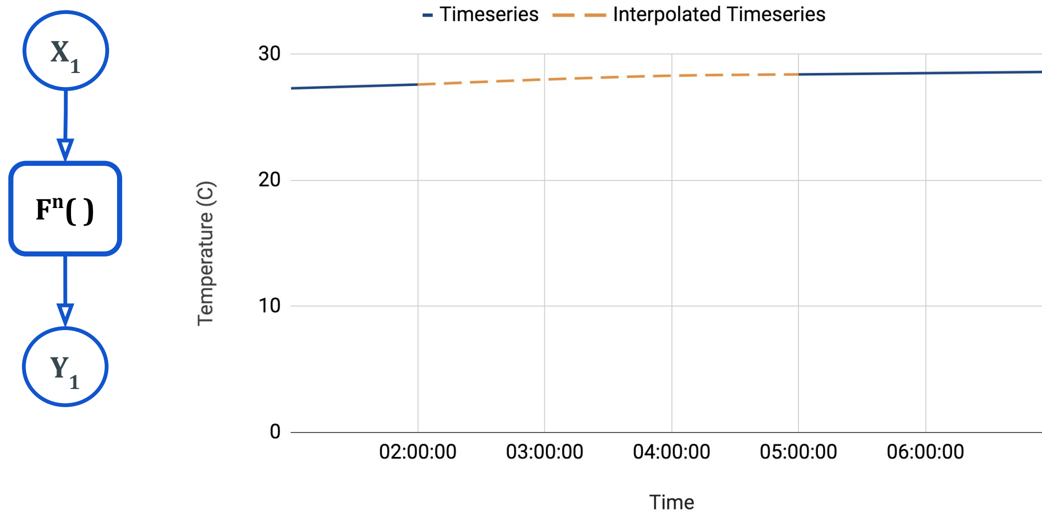

Figure 3.8: Serial interpolation.

Spatial Interpolation

During spatial interpolation, we use interpolation to either fill gaps in timeseries or create a new timeseries for a location using data from other (spatially distributed) locations' timeseries. Spatial interpolation can also be applied for sampling scalar timeseries from grid timeseries, for re-sampling grids, or for creating grids from timeseries data. For example, let us assume that there are precipitation timeseries data available for location A and location B as shown in Figure 3.9. Then we want to calculate the mean precipitation of nearby locations using these two timeseries. We can use the spatial interpolation to get the mean precipitation of nearby locations.

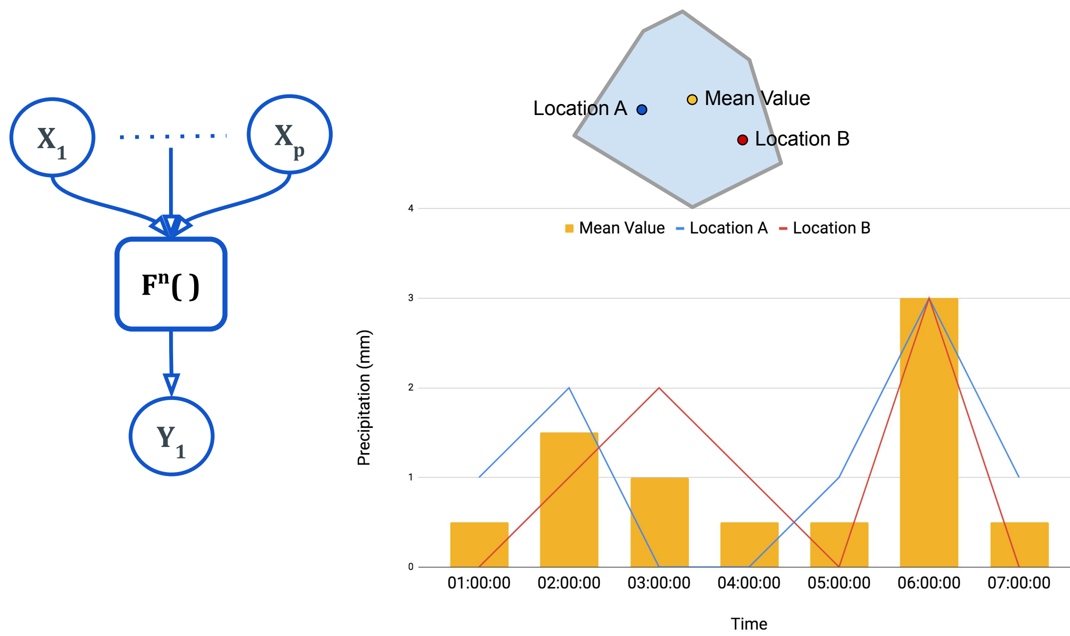

Figure 3.9: Spatial interpolation.

3.5.2 Transformation

Simple arithmetic manipulation of time interval transformation and shifting the series in time-specific hydro-meteorological transformation such as stage-discharge relationships are considered transformation. For example, as seen in Figure 3.10, there can be a timeseries of precipitation with 1-minute sample data. There will be a model requirement that needs hourly data. Therefore, we need to accumulation data manipulation to create a timeseries of hourly data using 1-minute sample data.

3.5.3 Validation

Validation functions check for counting reliable, doubtful, unreliable, and missing values. The weather stations can produce invalid observations due to technical errors in the sensors of the devices. If we feed those values directly to the models, the models will produce inaccurate forecasting. To avoid that, we need to modify or flag those values before feeding them to the numerical models. As seen in Figure 3.11 We can use validation extensions to modify or flag invalid data. For example, a weather station can record temperature above 70 celsius due to sensor error or calibration issues. However,

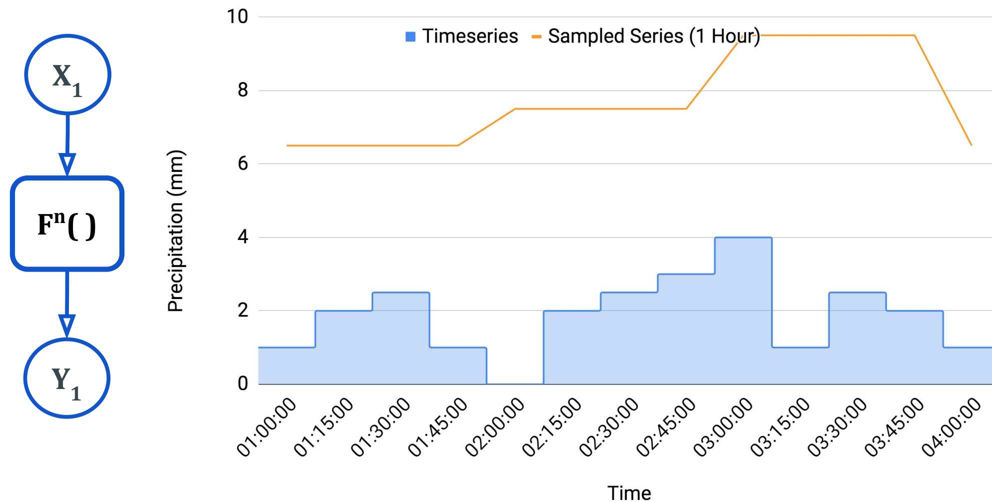

Figure 3.10: Time interval aggregation.

in the real world, it is not possible to have such a temperature in nature. Thus, we need to flag such temperature issues and repair the weather station as soon as possible.

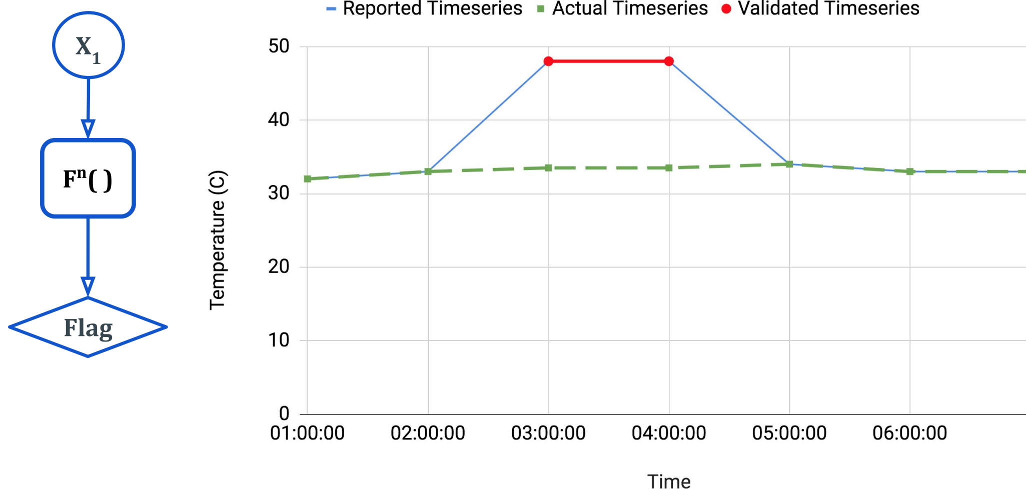

Figure 3.11: Timeseries data validation.

3.5.4 Extension API for Data Prepossessing

We exposed an API to create new triggers for extensions via /extension endpoints. Following is the accepted payload of the request while creating a new trigger. Later

we discussed the API in detail and how we can use the API to create new triggers to use extension modules.

```
{
    "extensionId": "",
    "extension": "Interpolation/Transformation/Validation",
    "function": "",
    "variables": [
        {
            "variableId": "",
            "metadata/metadtaIds": {
            }
        }
    ],
    "inputVariables": [],
    "outputVariables": [],
    "trigger": [
        {
            "trigger_type": "OnChange/OnTime",
            "trigger_on": []
        }
    ],
    "options": {
    }
}
```

- *extensionId* – An unique identifier for new extension trigger. Should be unique among all extensions.

- *extension* – The main extension category which is responsible for handling the extension with extracting data required and storing the output. Interpolation/-Transformation/Validation support for the moment.

- *function* – Name of the microservice which is handling the input to output mapping. `Fn()` function in the Figure 3.7.

- *variables* – Array timeseries mapping to variables. Multiple variables can be define with metadata or metadataIds of timeseries. Can be use same variable on inputVariables and outputVariables below.

- *inputVariables* – Timeseries that need to be provide into the function.

- *outputVariables* – Timeseries that output from the function.

- *trigger* – Type of trigger for the extension. Should be one of OnTime or OnChange.
  - *OnChange*: *trigger_on* — List of timeseries to listen on change and trigger the extension.
  - *OnTime*: *trigger_on* – List of schedules that need to be trigger the extension in cronjob string format.
- *options* – Bind Constant which is bind at the time of creating new extension trigger and pass on triggering the extension.

Extension Handler

Extension handler is responsible for triggering the correct extension with timeseries metadata when there is any change in the subscribed timeseries which are provided as a list of *OnChange* timeseries while creating an extension trigger. When new data is imported into one of the scalar adapter or vector adapter, those microservices notify the extension handler. Then the extension handler checks whether there is any extension subscribed for imported timeseries. If there is an extension needed to be triggered, then it invokes the given extension with the required data.

To improve the performance of matching triggers against a timeseries, we used two mechanisms as below. First, other than only storing extension metadata in a RDBMS instance of extension adapter, we also store extension data in a reverse lookup table called triggers while creating new extension triggers. Using this reverse lookup table, we can retrieve the extensions against a given timeseries without heavy processing and less delay. Second, we cache the resulting trigger type and timeseries metadata using the Redis database inside the extension handler for fast access and avoid too many queries on the RDBMS database. For the separation of service responsibilities, the extension data stored in the persistence database of the extension adapter and exposed via an endpoint.

3.5.5 Extension Scheduler

Extension scheduler is responsible for triggering the correct extension with timeseries metadata at a given time that provided while creating an extension trigger as *On-Time* cron job values. To improve the performance of triggering cron jobs at given time schedules, we used a few optimization mechanisms as described below. Same as for extension handler, the extension adapter exposed an endpoint to retrieve extension metadata grouped by cronjob time. Then we create cron jobs as per the data retrieved via extension adapter. We used a low-level programming language to simulate the cron jobs for fast processing the extensions. After getting extension triggers, the extension handler grouped them based on cronjob time and scheduled cron jobs programmatically to trigger at given times. When each cron job triggers at the scheduled time, each cron job also contains the extensions to be triggered. Fetching extension triggers data, and creating cron jobs occur with a long period to avoid huge overhead on extension adapter. When a user creates a new extension trigger, those data push into a queue in the Redis database. Then the extension handler pulls data from the queue in regular time intervals and creates new cron jobs for mentioned cronjob time. As mentioned above, to avoid the expansion of the programmatic cron job list that we created for new extension triggers, all cronjobs fetch again after a long period and flush the previously scheduled cron jobs. While doing that, we also flush the Redis queue. The extension handler should be run as a single process for the atomicity, to avoid triggering the same extension twice.

3.6 Query Timeseries

We used the query adapter primarily for supporting geo-spatial queries. As the primary service for searching for timeseries, it supports retrieval of timeseries metadata for external queries as well. Appendix B described the search query capabilities within WDIAS.

The system provides timeseries metadata query endpoints to search for existing timeseries metadata within the system using primary key attributes mentioned in the

Section 3.4.1. Also, it provides a set of geo-query endpoints, and users can perform geo-search queries with sending geo-information in geoJson format. The following geo-queries are supported by the system, and a detailed description is presented in Appendix B.

- Search for available locations within the provided geoJson area in the query.
- Search for available parameters in provided locations in the query.
- Search for available timeseries for a provided location in the query.
- Search for available timeseries for the given locations in the query.
- Search for available timeseries for given parameter. E.g., get available waterlevel timeseries in the system.
- Search for available timeseries within the provided geoJson area in the query.
- Search for available timeseries within the provided geoJson area by parameter. E.g., get available waterlevel timeseries within a given area.
- Query for all timeseries in the system.

3.7 Summary

Throughout the Chapter 3, we discussed the methodology of designing and implementing WDIAS based on the concepts studied in the literature review of Chapter 2. We used microservice architecture over SOA with an ESB to avoid the drawback of streaming bulk data through an ESB. To take the full advantage of using the microservices architecture, we moved to container orchestration based architecture rather than building tightly coupled microservices with the actor model. While implementing the import and export modules, those modules followed the microservice architecture style of *smart endpoints and dumb pipes*. Each microservice is implemented as a containerized application that can deploy independent of other microservices. Also, we used the microservice architecture pattern of *database per service* by combining different

types of database systems to create the WDIAS database structure. Further, we provided a comprehensive extension module system that allows users to preprocess data by creating new triggers without any downtime for system configuration. We used a geo indexing mechanism to support geo-search queries and indexed the timeseries metadata to support the search for any timeseries in WDIAS.

Chapter 4

Performance Analysis

This chapter includes the performance study of the WDIAS. Section 4.1 describes the test plans for the performance testing. Section 4.2 describes the logic behind creating the workload to do the performance testing planned in Section 4.1. Section 4.3 includes the results observed during performance study and summarised the observations in Section 4.4.

4.1 Test Plan

In Section 3.1, we discussed three modules of weather data integration and assimilation systems such as integration, assimilation, and dissemination. We can map each of those modules to WDIAS modules, which provide the same functionalities as below.

1. *Import* modules - Integration modules

2. *Export* modules - Dissemination modules

3. *Extension* modules - Assimilation modules

Other than the above modules, *adapter modules* act as the core modules and the base of the system. Also, the *query module* allows the users to query for the timeseries metadata and perform geo-based queries based on the locations.

The test plan is to perform load testing on the whole system and analyze the scalability of the system while measuring the operations performed on each module. Two

variables can vary while doing performance testing.

1. Number of requests made to the system at a given time i.e., Requests Per Second (RPS)

2. Request size

A timeseries is a series of data points in time order. The system may receive the data points of a timeseries in multiple requests. The request size can vary based on the use case from one data point to multiple data points. For example;

- A weather station may send the current water level in a regular 1 minute intervals. In such a case, it always comes as a request with one data point with a higher frequency.

- Due to resource issues such as batteries, a weather station can collect data points for a given period and send as a batch. An example scenario would be, a weather station can collect the precipitation and send all the data as a batch.

- The forecast precipitation data from the WRF model extract for a few days in hourly intervals and store them.

The percentage of number of requests per each data type that comes to the system at a given time can be varied depending on the situation. The majority of weather data requests come from weather stations and other observation tools. Then we need to validate and modify those timeseries data and store them as new timeseries that can directly feed into the NWMs. These weather stations can send the weather data at different frequencies with different parameters such as precipitation, temperature, humidity, and wind direction. Most of the timeseries mentioned above are scalar time-series. When compared to scalar data, parameters such as wind direction are vector timeseries with a lesser percentage of requests. Alternatively, most of the grid type data was generated as results of NWMs. But those grid data produce in less frequency due to the cost of the computer resource to produce results. After considering those factors we came up with 70% scalar, 20% vector, and 10% grid percentage values as

the expected requests rate ratios. As an upper limit of such scenarios while designing the performance testing, it uses the following;

- Hourly (24 data points)
- Every 30 minutes (48 data points)
- Every 15 minutes (96 data points) - One request is equivalent to a set of data points sent by a weather station in 15-minute interval for a single day

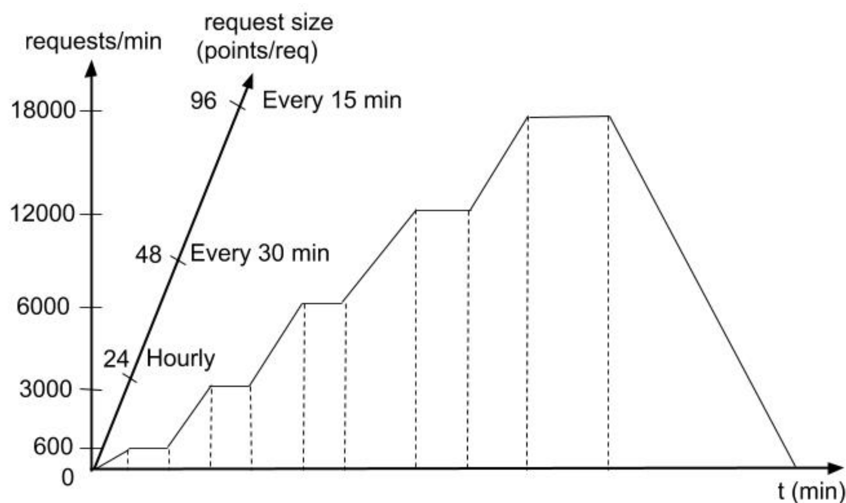

Figure 4.1: WDIAS load testing plan with changing the request size and RPS.

Figure 4.1 shows the summary of the load testing plan with changing the request size and performing load testing with increasing the RPS over time. As shown in the figure, for a given request size, the RPS increased in steps over the elapsed time. In each step, the RPS holds for a moment to give some time for the system to get stabilized. At each step, we can measure the performance of the system and study the stability of the system. After the test plan comes to the peak RPS, the system holds for more time than the previous steps. By holding the peak load for some time, we are planning to show the system is capable of scaling more than the proposed peak load in the testing plan. During the WDIAS load testing, we are planning to have a peak of 300 requests per second at peak time. After the peak RPS is over, the test case goes through a cool-down period to measure the ability to shrink down the system resources when there is not any load on the system.

4.1.1 List of Test Plans

Following are the test cases which are planned to performance test cases;

1. Test setup to create timeseries in 1-hour, 30-minute and 15-minute intervals, create metadata of the timeseries and create random extension triggers
2. Import + Extension + Export + Timeseries queries
   - Plan for 30 minutes of the test run with the request size of 1-hour (60-minute) data
   - Plan for 30 minutes of the test run with the request size of 30-minute data
   - Plan for 30 minutes of the test run with the request size of 15-minute data
   - Plan for 30 minutes of the test run with the request size of 15-minute data with auto-scaling
3. Query
   - Query timeseries
   - Plan for the 5 minutes of test run

4.1.2 Performance Metrics

During the performance analysis, we used the following metrics to measure the performance of the system.

- *Throughput* – Number of requests that can be processed per unit time. Normally for WDIAS performance observations, it uses Requests Per Second (RPS) to measure the throughput.
- *Latency* – Taken time to respond to the request sent to the system. Normally we measure with milliseconds (ms) in the WDIAS. During the performance analysis, we measure the minimum response time, maximum response time, the average response time, and standard deviation. The standard deviation measures the mean distance of the values to their average value. It gives an idea of the

dispersion or variability of the measures to their mean value. So, if the deviation value is low compared to the mean value, it will indicate that the measures are not dispersed (or mostly close to the mean value), and the mean value is significant. Other than that, we measure the response time into 90% percentile while giving the best response time up to 90% percentile of the responses.

- *Resource utilization* – Resource utilization is calculated based on the CPU and Memory usage via the K8s metrics server. It calculates the resource utilization over a 60 seconds time window. Network usage is getting based on the actual physical machines.
- *Auto-scaling* – We performed the load testing with two system configurations. First, we ran the microservices with a fixed number of pods inside the K8s cluster. Then k8s are only allowed to vertically scale each microservice until a container hits the CPU and memory limits defined per container or overwhelm the physical node resources. Secondly, we ran the load testing with auto-scaling enabled only for microservices, which are using higher resources with the above system configurations. Auto-scaling spawns new copies of the same microservice when the microservice gets more load than the configured percentage, and shrink the running instances when there is a lesser load.

4.2 Workload Generation

As explained in the research methodology (in Section 3.4.1), we selected six primary attributes from the timeseries metadata while implementing the system architecture. Before storing the data in WDIAS, we need to register timeseries metadata for the timeseries, and are required to provide values for the six attributes. During the test for location attribute, we selected set of locations all over the world as a generic case to include all the locations without binding to a specific area.

We used 250 locations from Googles countries public data set [27], and considered static throughout the testing. For each location, we created four timeseries metadata by using four different values for the moduleId key attribute. Thus, combining all

together first, we created 1000 timeseries metadata during the performance test and then stored the timeseries data against timeseries unique identifiers. We have written automated scripts to generate these data for the performance testing, and scripts are generic enough to create more timeseries metadata by changing the configurations.

For storing grid data, we used static 100 grid locations. Since grid locations have different metadata structures, we had to generate metadata for them separately. However, the locationID of the grid timeseries metadata is similar to scalar timeseries metadata that have the same effect when defining timeseries metadata.

We used the Apache JMeter 5.0 version [28] for performing the load test since it is a widely using open source tool which supports distributed load testing. As mentioned in the JMeter best practices, ”If your test needs large amounts of data - particularly if it needs to be *randomized*, create the test data in a file that can be read with CSV data set. This avoids wasting resources at run-time”. While running the load test, JMeter iterates over the timeseries metadata CSV file and assigns a new thread to work on the test scenario with given timeseries metadata. Instead of providing the randomness at run time, we stored the timeseries metadata in random order that satisfies the percentage of 70% scalar, 20% vector, and 10% grid, as mentioned in Section 4.1. Before running the test cases, we generated the timeseries metadata in random order and wrote them into a CSV file during the setup phase. The metadata CSV file arranges in a way that, within every ten lines, seven lines are scalar timeseries, two lines are vector timeseries, and 1 line is a grid timeseries.

WDIAS is capable of storing any numeric value up to three decimal points as per the default configuration. Even the parameter type is either precipitation, temperature does not make any difference, because in storage perspective, we are storing numeric value. Thus, during the performance testing, we only use real precipitation data from CUrW-SL. So, we used data from five weather stations for a month while preparing the test data for scalar timeseries. We used a script to clean up and prepare the precipitation data, and it is available as setup precipitation in WDIAS codebase.

Listing 4.1: Preparation of precipitation data.

```
1  Set ROOT_DIR=~/wdias/wdias-performance-test
2  -h | --help: Usage
3    setup_precipitation.sh  <COMMAND>
4      - COMMAND: help | extract | prepare | cleanup | populate
5    e.g.
6    setup_precipitation.sh prepare
7      Segregate single file data into multiple files based on date.
       And separate into main directories of 15min, 30min, 60min and
       create tar files
8    setup_precipitation.sh extract
9      Extract the tar files into 15min, 30min and 60min
10   setup_precipitation.sh cleanup
11     Clean up extracted directories
```

While preparing the data, we created three sets of timeseries based on the time interval, such as hourly (60-minute), 30-minute, and 15-minute data, with modifying the replicated data. The above method reduced the overhead of filtering data during performance testing. We also set up the JMeter as a containerized application, which allows us to run inside the K8s cluster. As an optimization for the JMeter container creation, we avoid adding extensive test timeseries data, which causes the container size to be large and making the build process slower. Rather than doing that, we used extract scripts while running the container and extract the data into the container. For each test plan, we used a date counter and increment the date after one full cycle of timeseries metadata CSV file is processed. When the date counter increases, it reads the real data for that day. After the date counter advances to the next month, we repeat reading data from the beginning of the month. To provide more randomness, we switch over five locations during the date increment. The JMeter reads the data via the JSR223 scripts, and we have written those scripts in a generalized style such that it is possible to tweak and use if there are more location data available.

Similar to the scalar test data prepared, we prepared the test grid data using the real water-level output of the FLO2D 150 meter hydrology model that is used in the CUrW-SL. The FLO2D timeseries are produced as ASCII Grid files, and each data point represents a grid file that consists of 120 rows and 139 columns. To represent missing cell values of the ASCII grid file, we used the `-9` instead of `-999` as an optimization. Even with that, each file size is around 50,337 Bytes (50 Kilobytes).

Listing 4.2: Preparation of water level data.

```
1 Set ROOT_DIR=~/wdias/wdias-performance-test
2 -h | --help: Usage
3   setup_water-level.sh  <COMMAND>
4     - COMMAND: help | extract | prepare | cleanup | populate
5   e.g.
6   setup_water-level.sh prepare
7     Segregate single file data into multiple grid file
     directories based on date. And separate into main directories
     of 15min, 30min, 60min and create tar files
8   setup_water-level.sh extract
9     Extract the tar files into 15min, 30min and 60min
10  setup_water-level.sh cleanup
11    Clean up extracted directories
```

We also used the real water level data for one month period and developed a script to do the data preparation, cleanup, and extract. Similar to the scalar test scenario, the JMeter repeats reading from the beginning of test data when the date counter increments to the next month.

4.2.1 Experimental Setup

To create a higher workload, we used JMeter's distributed testing capabilities. JMeter uses the master-slave approach for distributed testing and allows us to handle all the test cases via a single master instance. However, it has the limitation of *the same test plan runs by all the servers* rather than distributing the work among slave instances. In other words, JMeter does not distribute the load between servers, and each slave service runs the full test plan. So, if we set 1,000 threads per test plan and run the test using six JMeter servers, the distributed test setup ends up injecting 6,000 threads. During JMeter distributed mode, the master server triggers the same copy of the test plan on configured slave nodes in parallel. After successfully running the test plans, the master server gathers results from slave servers.

While implementing JMeter test plans, it uses threads to simulate the users. The root component of a test plan is the thread group. There are different kinds of thread groups available such as thread group (basic thread group), arrivals thread group, free form arrivals thread group, stepping thread group, concurrency thread group, and ultimate thread group.

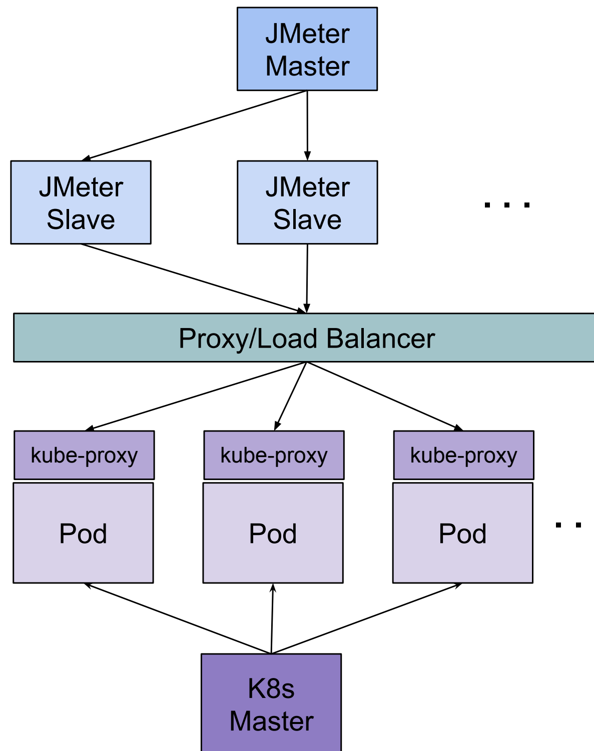

Figure 4.2: Experimental setup with Apache JMeter.

Closed vs. Open Workload Models

- *closed system model* [9] – A new request is only triggered by the completion of a previous request, following by a think time. The system has negative feedback that makes it impossible to bury-out the service, so users wait for the responses before making new requests.

- *open system model* – New requests arrival independently of completions, e.g., according to a stochastic process or fixed trace. The system has no negative feedback.

While implementing the test cases for WDIAS, we used the concurrency thread group with the throughput shaping timer extension of JMeter because it supports the open workload approach. Using other thread groups, we are required to find the exact number of threads and timer delays that produce the desired number of RPS to the server, which is known as the *closed workload*. By using the *throughput shaping*

timer, we only need to configure the RPS throughout the test plan. Then the throughput shaping timer provides a feedback loop to the concurrency thread group via schedule feedback function to dynamically maintain thread count required to achieve target RPS [29].

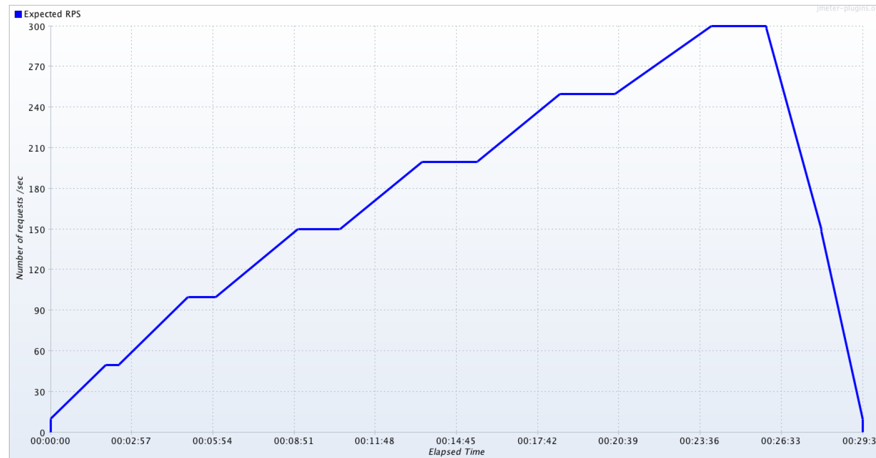

Figure 4.3: WDIAS load testing throughput shaping timer RPS configurations.

Figure 4.3 shows the throughput shaping timer configurations that we used for the load testing, and it is similar to the load test plan described in Section 4.1. The maximum peak load that can be had during the load test is 300 RPS (equivalent to 18,000 requests per minute). The followings are the test plans that we used during the performance testing:

- User-defined variables such as for change variable size
- Setup thread group for test setup before load testing
- Create timeseries thread group for create timeseries metadata before import and export timeseries data
- Create extensions thread group for create triggers for basic extensions
- Load test timeseries with concurrency thread group

The first step of the test plan consists of setting up the timeseries metadata. At the same time, we create the timeseries metadata in the WDIAS system. Even with the real

use case, users only have to register timeseries metadata once and reuse the generated timeseries identifier to insert the data. The following list of test steps shows the test plan for setting up the timeseries metadata.

- Load the timeseries metadata CSV
- Create timeseries
- Check with get timeseries
- Query for timeseries

By creating the timeseries metadata before, we can reduce the overhead during the performance testing with only focus on essential data storing, and cause more accurate RPS at the run time. Otherwise, JMeter needs to allocate another set of threads for registering timeseries metadata. Further, when we consider real use cases, most of the time, the users want to search for timeseries metadata and retrieve relevant metadata, rather than registering new timeseries metadata in the system.

The following list shows the test steps of the JMeter test plan for all modules while performing the load testing.

- Load the timeseries metadata CSV
- If scalar or vector then insert relevant timeseries data, otherwise insert grid timeseries data
- Wait for extensions timeseries
- Query for timeseries metadata
- If scalar or vector then retrieve timeseries data, otherwise retrieve grid timeseries data
- Increment date

As mentioned in Section 4.1, we performed another separate test plan on the query timeseries module. Here, we used the following test steps to cover most of the timeseries metadata search queries and geo-queries supported by the WDIAS.

- Load the locations CSV
- Prepare interest area for search for the timeseries
  - /Location: * → Locations
  - /Location: Area → Locations
  - /Parameter: Location → Parameters
  - /Parameter: Locations → Parameters
  - /Timeseries: Location → Timeseries
  - /Timeseries: Locations → Timeseries
  - /Timeseries: Locations, Parameter → Timeseries
  - /Timeseries: Area → Timeseries
  - /Timeseries: Area, Parameter → Timeseries
  - /Timeseries: *, Parameter → Timeseries
  - /Timeseries: * → Timeseries

With the above JMeter test plans, we have to enable each test plan separately and before running with JMeter distributed mode. To make the process faster, we implemented a script tool to interact with the JMeter test stored files. The script is available within the code base [30] as *test-dev* and supports the following features.

Listing 4.3: Automated performance test plans.

```
1  -h | --help: Usage
2    test-dev enable <MODULE>
3      - MODULE: import(i) | export(e) | extension(x) | all(a)
4
5    test-dev run <REQ_SIZE>
6      First need to enable module that need to be run
7      - REQ_SIZE: 24(1) | 288(2) | 1440(3)
8      NOTE: Modify test.conf as necessary
9      e.g.
10     test-dev run 24
11     or
12     test-dev run 1
13
14   test-dev once <REQ_SIZE> <SEARCH_PHASE>
15     This will enable the test case first. Then run the test case,
       and at the end disable and exit.
```

```
16      - SEARCH_PHASE: Thread Group level name that matches
17      e.g.
18      test-dev once 24 CreateExtensions
19      or
20      test-dev once 1 CreateExtensions
21
22    test-dev disable <MODULE>
23      - MODULE: import(i) | export(e) | extension(x) | all(a)
```

JMeter performance tuning

- Use CSV files rather than random number generation
- Use JSR223 scripts with Groovy for data processing for test cases
- Using throughput shaping timer to get more accurate RPS

4.2.2 Configure Experimental Setup on Cloud

As we described in the Section 3.3, the WDIAS is mainly based on microservice architecture and implemented on top of Kubernetes. Kubernetes (K8s) is an open-source system that is capable of automating deployment, scaling, and management of containerized applications. During the performance study, we used Amazon Elastic Kubernetes Service (Amazon EKS). There are many K8s solutions available such as GCP, Azure, and Digital Ocean on the cloud other than Amazon EKS.

We have documented a detailed description of setting up the Amazon EKS [31], and it is available in the codebase of the WDIAS. K8s is a planet scaling tool [32], and users can tweak the configuration to run the WDIAS as per the required level of load on the system. We used a set of physical nodes, as mentioned in Table 4.1, to set up the K8s for the performance testing with a peak of 300 RPS.

Table 4.1: Amazon EKS nodes

Node Label	vCPU	RAM (GB)	Storage (GB)	Quantity	EC2 Name
core	16	32	15	1	c5.4xlarge
grid	8	16	25	1	c5.2xlarge
scalar	8	16	20	1	c5.2xlarge
test	4	10.5	5	1	c5n.xlarge

The WDIAS uses Helm [33] which is a package manager for K8s. Each microservice within the system deploys and maintains as a Helm chart and the Helm charts [34] for each microservice available within the codebase. For the databases, we used the official helm charts. Using Helm, we can configure resources, the number of replicas needed, deploy new changes, and manage credentials, storage, and proxy settings that are needed for each microservice.

4.2.3 Performance Tuning

According to Table 4.1, we selected a few physical nodes with different resource capacities. The grid and test labeled nodes have higher network bandwidth. To get better performance using the available resources, we can schedule each microservice into a predefined set of nodes. Thus, we tried to deploy all JMeter microservices on test labeled nodes since the test plans need to interact with all the services and transfer bulk timeseries data. Also, the grid microservices are hosted inside the grid labeled node due to the massive size of grid data timeseries that need to be transferred during the load testing. There are few ways to assign microservice to a particular physical node in the K8s;

- *nodeSelector* – Hard rules match for node schedule. If not match, pods will not schedule.
- *node Affinity and Anti-Affinity* – Soft rules for node schedule. Based on the rules, pods will schedule on the most appropriate node.
- *Taints and Tolerations* – Opposite of nodeSelector, repel mismatching pods away from the nodes.

Users can use one of the above methods to schedule microservice on physical nodes for better performance. For the moment, we used node affinity to schedule the microservices. We chose the above method since it allows us to schedule all the microservices in a node when there is only a single physical node available. Microservices are assigned to each node in Table 4.1 based on the node label as below;

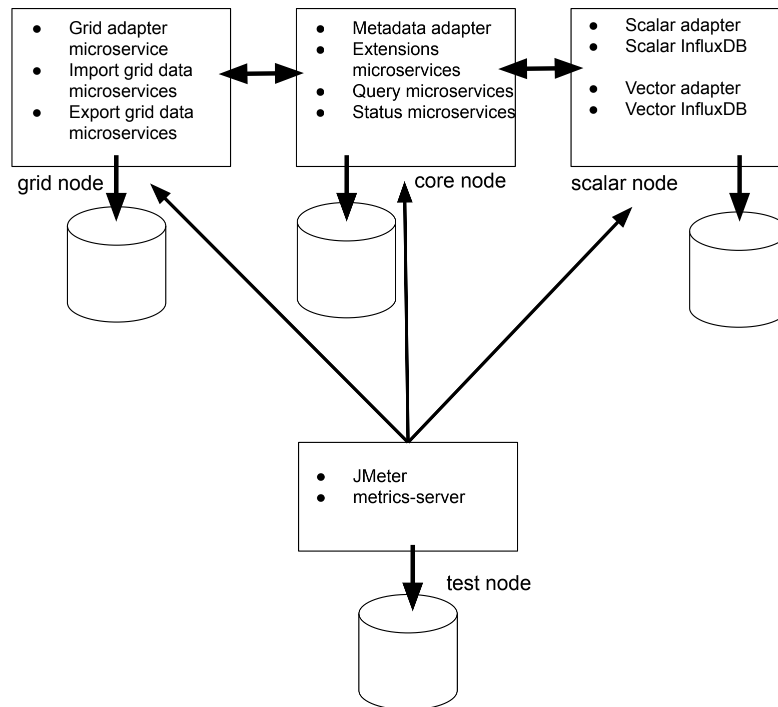

Figure 4.4: Amazon Elastic Kubernetes Service (Amazon EKS) node setup.

- *core* – metadata, extensions, query, status and extension services
- *grid* – grid adapter, import and export grid data
- *scalar* – scalar and vector adapters, import and export of same data types
- *test* – JMeter and metric server

Figure 4.4 shows the overview of the microservices assignment to each node in the Amazon EKS setup. Further, it shows the connectivity of the nodes and the data interactions via the arrows. We scheduled the microservices in the core labeled node together, since most of those services are communicating together, and running them inside one physical node reduces the latency. Then we separate scalar and vector microservices and their databases to separate the load from the core node while storing scalar and vector timeseries data. Similarly, we assigned microservices to grid and test nodes based on the higher bandwidth requirements.

4.3 Performance Test Observations

This section includes the observations based on the test plans discussed in Section 4.1, and the experimental system setup explained in Section 4.2. Sections 4.3.1 to 4.3.4, we discuss the observations collected with performing the load test plans with 60-minute data (24 data points per request), 30-minute data (48 data points per request), and 15-minute data (96 data points per request) by varying the request size for scalar and vector timeseries. For grid timeseries, we performed the load test with ASCII grid file as data for each time step (with the size of 52 KB). On average, each sample grid data for 60-minute data requests consist of 1.2 MB, 30-minute data requests consist of 2.4 MB, and 15-minute data requests consist of 4.9 MB. Then we perform another round of 15-minute data with enabling auto-scaling for the higher resource utilized microservices.

4.3.1 Load Testing with Hourly Resolution Data

After running the test plan with 60-minute resolution data (24 data points per scalar and vector requests, and 24 ASCII grid files per grid requests), we observed the data summary of Table 4.2. The test plan performed 311×10^3 of sample requests within 30 minutes of elapsed time.

Table 4.2: Throughput and latency of load test with 60-minute data

Label	**Samples**	**Avg**	**Min**	**Max**	**90% Line**	**Std.Dev.**	**Error**	**RPS**
Insert Timeseries	71826	28	13	2773	31	58.74	0.00%	40.5
Retrieve Timeseries	71796	8	7	242	10	4.18	0.00%	40.7
Insert Grid	7982	23	21	126	26	4.23	0.06%	4.5
Retrieve Grid	7979	68	59	238	75	10.11	0.00%	4.5
Query: Location	71804	3	2	109	3	1.52	0.00%	40.5
TOTAL	311182	127	0	2773	503	207.80	0.00%	175.4

Insert timeseries performed over scalar and vector data types with each request sent with 24 data points. On average, it took 28 milliseconds to insert scalar and vector data into the WDIAS, and up to 90% percentile inserted data within 31 milliseconds. Noticeably it has 0% of errors, which means that all the requests completed successfully. Also, WDIAS handled 40.5 RPS of scalar and vector data requests on average with the

given workload. The retrieval of scalar and vector timeseries has lesser delay than the insertion such that retrieve on average 8 milliseconds, and up to 90% percentile retrieve data within 10 milliseconds. On the other hand, insert grid timeseries data performed within 23 milliseconds on average, and almost all the requests perform within the same amount of time. Hence, the standard deviation is smaller. Delay with inserting grid timeseries smaller because those requests are handled asynchronously. Retrieve grid timeseries data performed with a bit higher value of latency of 68 milliseconds on average due to performed the retrieve request on-demand. Both insert and retrieve grid timeseries data have 4.5 RPS on average since it only gets 10% of the number of requests from total number of requests during the load test. Location query test case runs for the scalar and vector timeseries metadata, and it was able to fulfill the requests within 3 milliseconds on average.

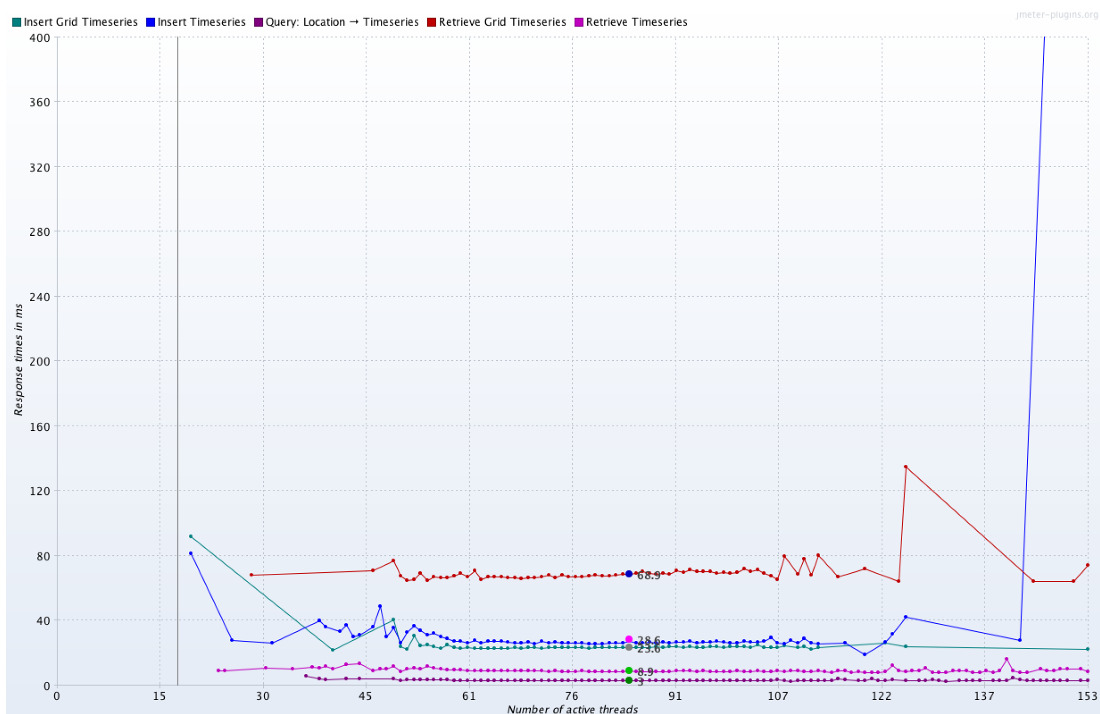

Figure 4.5: Response time vs threads while load testing with hourly data.

Figure 4.5 shows the response time against the number of active threads for 60-minute data requests. As the graph shows, the response time kept the same while increasing the number of active threads against each test case. The above graph shows the scalability of the WDIAS since the system is able to process more requests without

a significant change in the latency. When the number of active threads increased more than 122, Figure 4.5 shows uncertainty in the inserting and retrieving scalar and vector data. The performance degrades because those data handles on-demand and depends on the timeseries database performance. However, when the request sizes are smaller, WDIAS was able to perform insert and retrieval of grid data with the same latency.

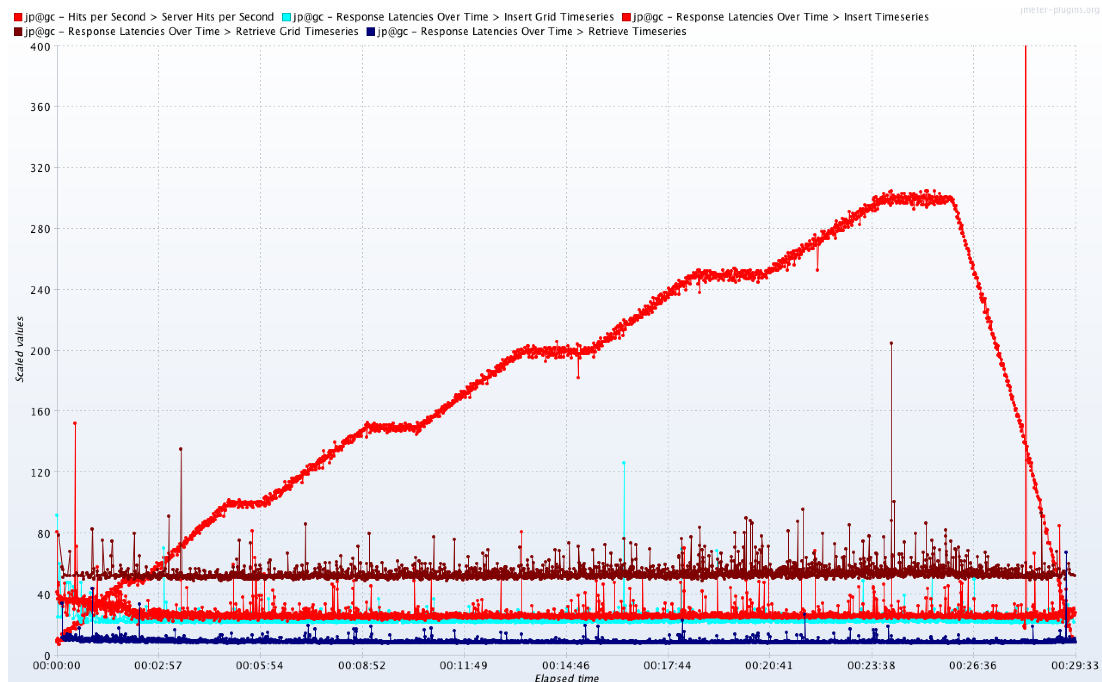

Figure 4.6: Latency against server hits while load testing with hourly data.

Figure 4.6 graph provides a better overview of the variation of latency over the elapsed time of the test plan against the number of server hits per second. This graph further proves that the system was able to handle the increasing workload without any significant change in the delay. The stepped red colour graph for server hits per second shows the number of requests that was sent by the load test with time. Other horizontal lines with spikes show the latency for each of test case, such as insertion and retrieval of scalar, vector, and grid data with time.

4.3.2 Load Testing with 30-minute Resolution Data

Next, we performed the test plan with 30-minute resolution data, which means 48 data points per each request for the scalar and vector data types, and 48 ASCII grid files

with 2.4 MB of size per each insertion request for the grid data type. Also, the test plan performed round up to 311×10^3 number of sample requests, which is almost similar to the number of samples processed with hourly resolution data. That means WDIAS-based system was able to process same number of request while increasing the request size with providing higher throughput.

Table 4.3: Throughput and latency of load test with 30-minute data

Label	Samples	Avg	Min	Max	90% Line	Std.Dev.	Error	RPS
Insert Timeseries	71759	29	14	1699	32	50.97	0.00%	40.5
Retrieve Timeseries	71730	9	7	1033	10	6.04	0.00%	40.6
Insert Grid	7972	44	40	162	49	8.17	0.08%	4.5
Retrieve Grid	7971	81	67	284	93	15.15	0.00%	4.5
Query: Location	71734	3	2	110	3	1.90	0.00%	40.5
TOTAL	310878	129	0	1699	503	207.10	0.00%	175.3

Table 4.3 shows the response latency summary details and RPS with 30-minute resolution data. The results are almost similar to the observations in Section 4.3.1. Even though we doubled the request size, WDIAS was able to keep the performance almost the same instead of increasing the latency twice. Insertion and retrieval of scalar and vector data increased by 1 millisecond on average. However, the grid data insert latency increased from 23 milliseconds to 40 milliseconds. Such doubling of latency is expected, as the grid data size was increased from 1.2 MB to 2.4 MB, and it is taking time to stream the data into WDIAS-based system. Retrieval of grid data also increased by 22 milliseconds, because during the test cases, we try to retrieve the inserted grid data in a different format, but the export data size also gets increased due to the increase of the insert grid data size.

Figure 4.7 shows the latency against the number of active threads for the test plan with 30-minute resolution data. As the graph shows, the response time kept almost constant while increasing the number of active threads. Thus, we can say that WDIAS is a scalable system since it was able to increase the throughput without a significant change in the latency. When the number of active threads increased, Figure 4.7 shows uncertainty in the insertion and retrieval of scalar and vector data. Some of the facts cause this is, those operations are handled on-demand. At the same time, the system is writing timeseries while reading from the timeseries database. The InfluxDB can

Figure 4.7: Response time vs threads while load testing with 30-minute of data.

fine-tune for writes or read based on the requirements, but we used it with default configurations. Also, the InfluxDB commercial version support horizontal database scaling with sharding, and we are using the basic open source support in this test setup.

Figure 4.8 graph provides a better overview of the variation of latency over the elapsed time of the test plan against the RPS. By referring to the graph, we can see that over the elapsed time the latency does not change very significantly. However, during the peak load, it shows a minor spikes in latency of the insertion and retrieval of grid timeseries. When compared to Figure 4.6, the latency of insertion grid timeseries data almost doubled. The grid timeseries data was handled asynchronously after successfully steamed the data into the WDIAS system, thus the latency for storing the data increased by a factor of two since the request data size also increased twice.

4.3.3 Load Testing with 15-minute Resolution Data

During this section, we performed the test plan with 15-minute resolution data, which means 96 data points per each request for the scalar and vector data types, and 96 ASCII grid files per each insertion request with the size of 4.9 MB for the grid data

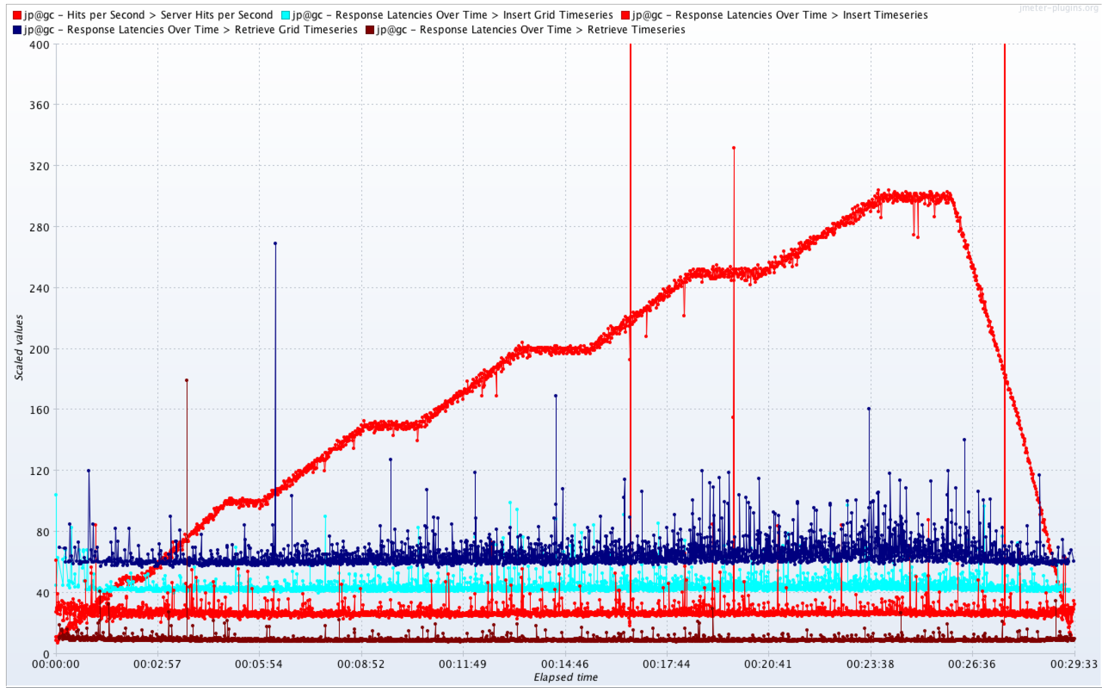

Figure 4.8: Latency against server hits while load testing with 30 minute of data.

type. The size of each request is four times larger than the 60-minute resolution data. Here also, the test plan performed approximately 311×10^3 of sample requests, which are almost similar to sample requests made during 60-minute and 30-minute resolution data. This means WDIAS-based system was able to increase the throughput while increasing load with the request size, and processed the same amount of requests within the elapsed time.

Table 4.4: Throughput and latency of load test with 15-minute data

Label	**Samples**	**Avg**	**Min**	**Max**	**90% Line**	**Std.Dev.**	**Error**	**RPS**
Insert Timeseries	71775	30	12	1719	41	51.71	0.00%	40.5
Retrieve Timeseries	71736	23	8	1623	32	50.18	0.00%	40.6
Insert Grid	7975	91	77	279	112	19.58	1.42%	4.5
Retrieve Grid	7972	118	80	876	165	56.15	0.00%	4.5
Query: Location	71749	3	2	130	4	2.32	0.00%	40.5
TOTAL	310934	134	0	1719	503	206.40	0.04%	175.4

Table 4.4 shows the response latency summary details and RPS of the test plan with 15-minute resolution data. When compared to the observations from the performance test with 30-minute data, the request latency also increased for all sample requests. The scalar and vector data insertion increased by 1 milliseconds, and retrieval of those

data increased significantly by doubling the latency. Also, the standard deviation of retrieval timeseries data increased by a considerable amount for all the data type's sample requests. One of the reasons for increasing the retrieval data delay was while performing heavy database writes on the InfluxDB database instance has effects on the database reads. The grid data insert and retrieve latency also doubled approximately because the stream data size increase by a factor of two compared to 30-minute data. Noticeably, the error rate of inserting grid data increased up to 1.42%. Even though we increased the request data size by four times higher, the system was able to handle all requests without significant performance issues. Also, the system was able to process same number of request as with 60-minute and 30-minute resolution data while increasing the throughput of the system with the request size.

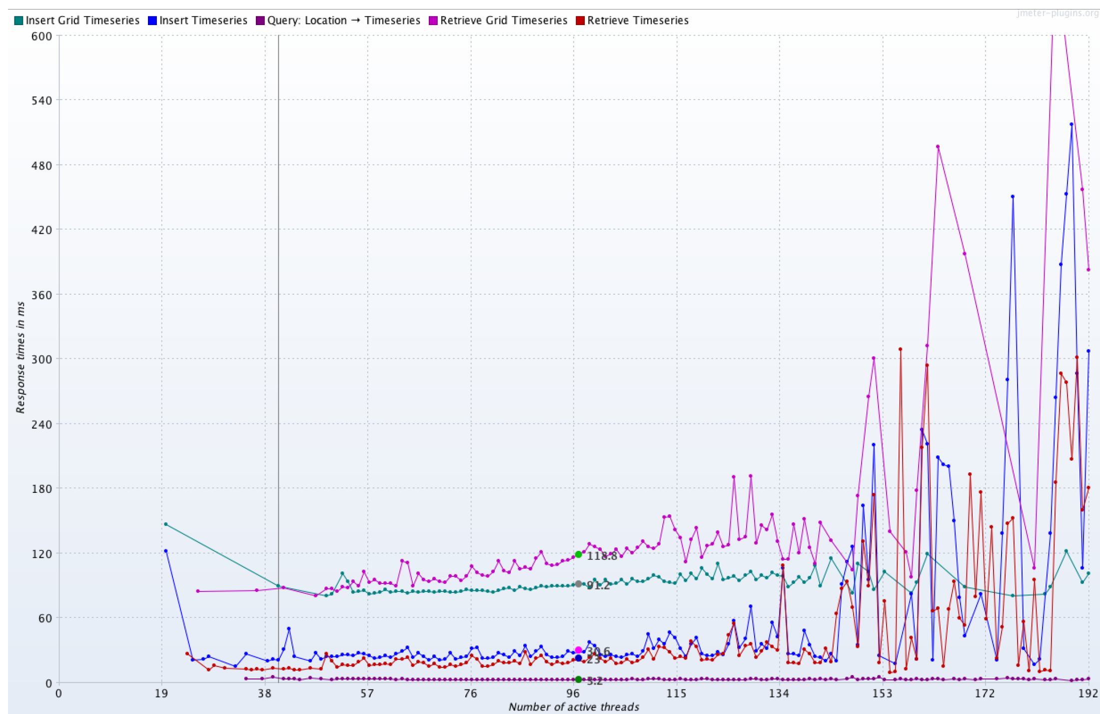

Figure 4.9: Response time vs active threads while load testing with 15-minute of data.

Figure 4.9 shows the latency against the number of active threads for the test plan with 15-minute resolution data. As per the graph, the system was able to keep the response latency constant while increasing the number of active threads. However, when the number of active threads became higher, there was a disturbance with the response delay during the insertion and retrieval of scalar and vector timeseries data.

As mentioned above, this may happened due to the system reaching its maximum capacity as per the allocated resources, or caused by heavy database read and writes on the InfluxDB free database version. The latency of insert and retrieval of grid timeseries data increased twice when compared to 30-minute resolution data. With a larger request size, we can notice the latency also increased by a smaller factor when we increased the number of active threads. When compared to the load testing with 60-minute and 30-minute data, we can see 192 active threads used by the JMeter to maintain the RPS. When we increase the request size, the latency also increases. Then the active threads have to wait for the response from the system rather than doing any other work. To maintain the server hits, the JMeter has to spawn more active threads.

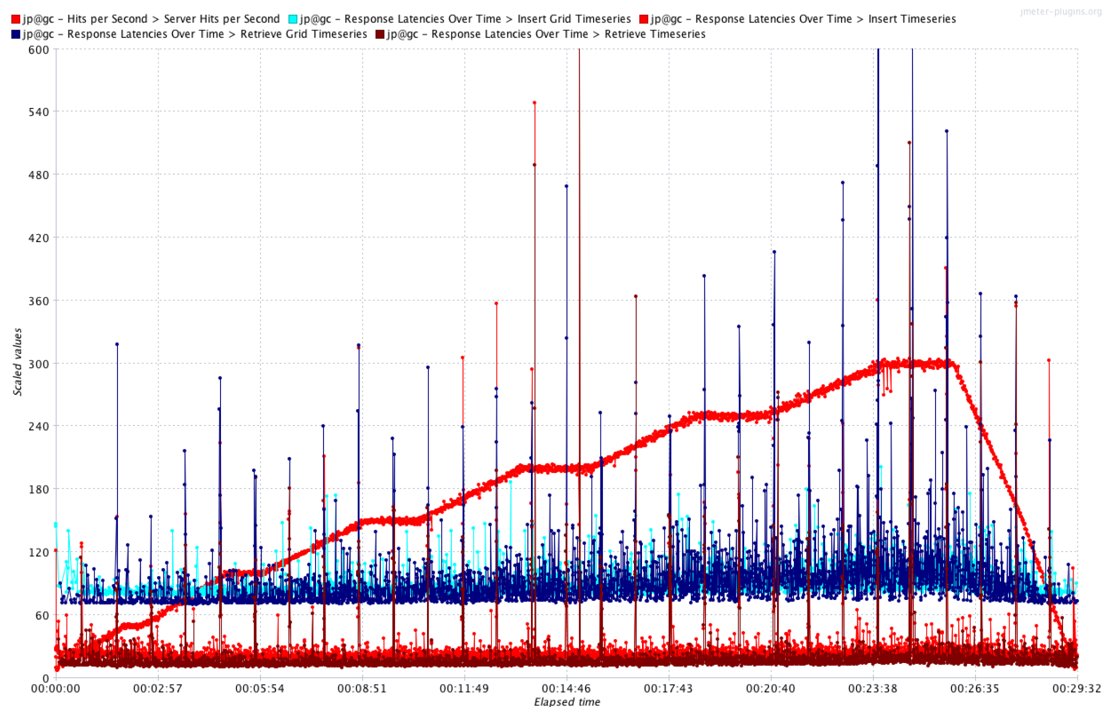

Figure 4.10: Latency against server hits while load testing with 15-minute of data.

Figure 4.10 graph provides an overview of the variation of latency over the elapsed time of the test plan against the number of server hits per second for 15-minute resolution data. As per the graph, the system was able to keep the latency constant all over the test plan. However, at the peak, the latency tends to vary from the mean value. Also, we can observe that there were lots of latency spikes throughout the test plan when we increased the request size four times.

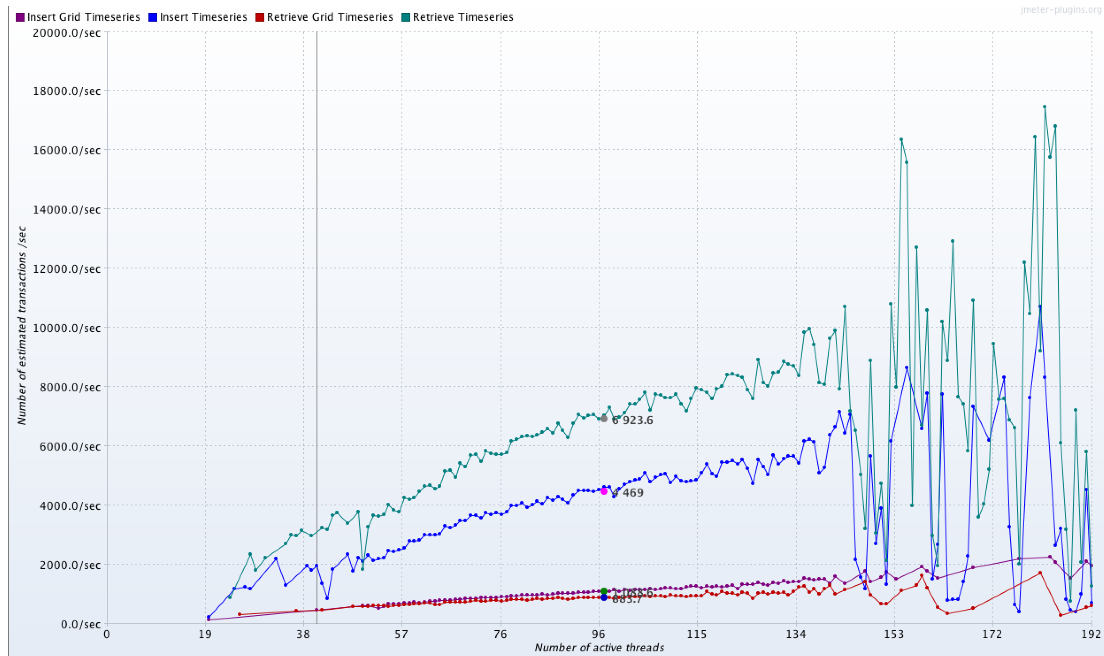

Figure 4.11: Transaction throughput vs thread while load testing with 15-minute of data.

Figure 4.11 shows the total server's transaction throughput against the number of active threads. The formula for total server transaction throughput is *active threads per second / one thread response time* [35]. It shows the statistical maximum possible number of transactions based on the number of active threads accessing the application at a given time. This estimates the transaction throughput based on the latency of requests against the number of active threads at a given time. We can assume that the highest number of active threads occur at the peak time of the load testing. However, the number of latency samples are limit with the higher number of active threads. Consequently, in Figure 4.11 we can see missing data points, and spikes with graph drawing with available data points. Further we can analyze this graph against Figure 4.9, and derive this output based on latency fluctuation in that graph. By combining the Figure 4.10, this graph shows that the throughput of the system gets increased without much change in the latency, thus it proves the scalability of the WDIAS.

4.3.4 Load Testing with 15-minute Resolution Data with Auto Pod Scaling

Here we did the performance test plan with 15-minute resolution data, which is similar to the above section. However, during the test plan, we enabled the K8s auto-scaling only for higher resource utilized microservices, as explained in Section 4.1.2. While performing the above test plans, we noticed the import-ascii-grid-upload microservice is using lots of CPU, and the adapter-grid microservice is using lots of memory. Before starting the test plan, we configured auto-scaling for the import-ascii-grid-upload microservices. However, the adapter-grid microservice is a consistent service for storing netCDF files. If we want to enable the auto-scaling for the service, all microservice instances of adapter-grid need to share data via shared storage. Since Amazon EKS does not support multiple reads and writes support volumes [36], we are unable to enable auto-scaling during the test plan. But some of other K8s platforms support multiple reads, and writes volumes. However, the test plan performed approximately 311×10^3 of almost similar sample requests during the above sections.

Table 4.5: Throughput and latency of load test with 15-minute data while enabled K8s auto-scaling

Label	**Samples**	**Avg**	**Min**	**Max**	**90% Line**	**Std.Dev.**	**Error**	**RPS**
Insert Timeseries	71727	34	13	1777	27	118.78	0.00%	40.5
Retrieve Timeseries	71693	7	5	1608	9	18.72	0.00%	40.5
Insert Grid	7968	87	77	233	98	14.07	0.18%	4.5
Retrieve Grid	7965	89	63	1694	110	37.79	0.00%	4.5
Query: Location	71704	1	0	203	2	2.05	0.00%	40.5
TOTAL	310734	130	0	1777	501	212.35	0.00%	175.3

Table 4.5 shows the response latency summary details and RPS as explained in Section 4.3.3. If we analyzed the performance of the auto-scaling enabled microservice, the average grid data insertion latency was reduced from 91 milliseconds to 87 milliseconds. Also, we can notice that the error percentage reduced from 1.42% to 0.18%. Since the standard deviation also reduced, we can conclude that the latencies are closer to the average latency. Further, these improvements in insert grid timeseries data seem to affect the output of retrieve grid timeseries data as well, and decrease the latency from 118 milliseconds to 89 milliseconds. When it comes to scalar and

vector timeseries data insertion, latency was increased a bit. However, we can notice that, the retrieval of scalar and vector timeseries latency improved from 23 milliseconds to 7 milliseconds. Specially, the latency for query location timeseries metadata was improved from 3 milliseconds to 2 milliseconds. It was constant with 3 milliseconds in previous load testings. Other than inserting scalar and vector data, other test cases standard deviation was improved, and we can conclude that the overall system performance is enhanced when auto-scaling is enabled.

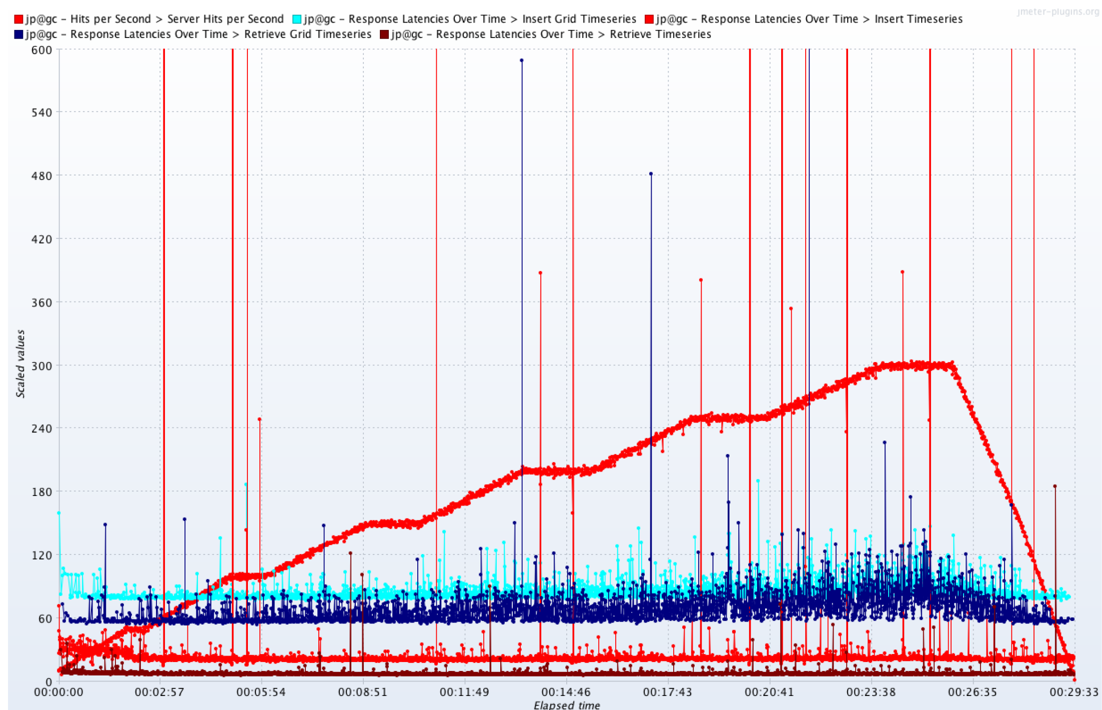

Figure 4.12: Latency against server hits while load testing with 15-minute of data with enabled auto-scaling.

Figure 4.12 graph provides an overview of the variation of latency over the elapsed time of the test plan against the number of server hits per second for 15-minute data while auto-scaling enabled for K8s. According to the graph, the WDIAS was able to keep the latency constant all over the elapsed time. When compared to Figure 4.10 without auto-scaling, we can see lesser latency spikes throughout the test period.

Figure 4.13 shows the total server's transaction throughput against the number of active threads. When compared to Figure 4.11 without auto-scaling, the estimated throughput with a higher number of active threads becomes stable, according to the

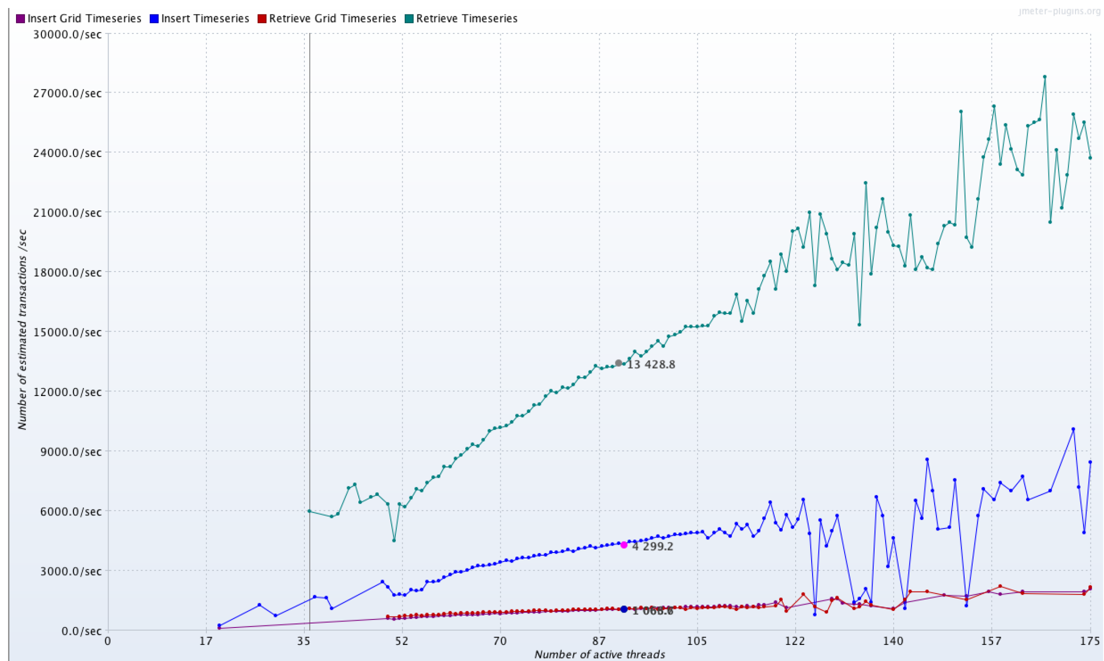

Figure 4.13: Transaction throughput vs threads while load testing with 15-minute of data with enabled auto-scaling.

graph. Thus, we can conclude that enabling auto-scaling also improves the throughput by supporting a higher number of active threads. We can see fluctuation when increase the number of active threads due to lack of sample latency with higher number of active threads during the peak of load testing. Further, we can see estimated throughput for insert and retrieve grid data also increased by a small factor, and it more smooth than 15-minute data load testing without auto-scaling. During the next paragraphs, we will discuss the resource usage while running the test plan while enabling the auto-scaling.

Figure 4.14 shows the CPU usage and memory usage for import timeseries modules and export timeseries modules with 1-minute resolution. We show the CPU usage with milli CPUs [37] (1 CPU = 1000m CPUs) and the memory usage shown with Megabytes (Mi) [37] in the graphs. Noticeably, the import-ascii-grid-upload microservice used around 10 CPUs at the peak time while using 3.6 Gigabytes (Gb) of memory. Another important fact is, after the test cases finished, the WDIAS cooled down the system via K8s auto-scaling and released the resources. Thus, we can see the elasticity of the WDIAS according to the workload.

Figure 4.15 shows the number of import-ascii-grid-upload pods scheduled overtime

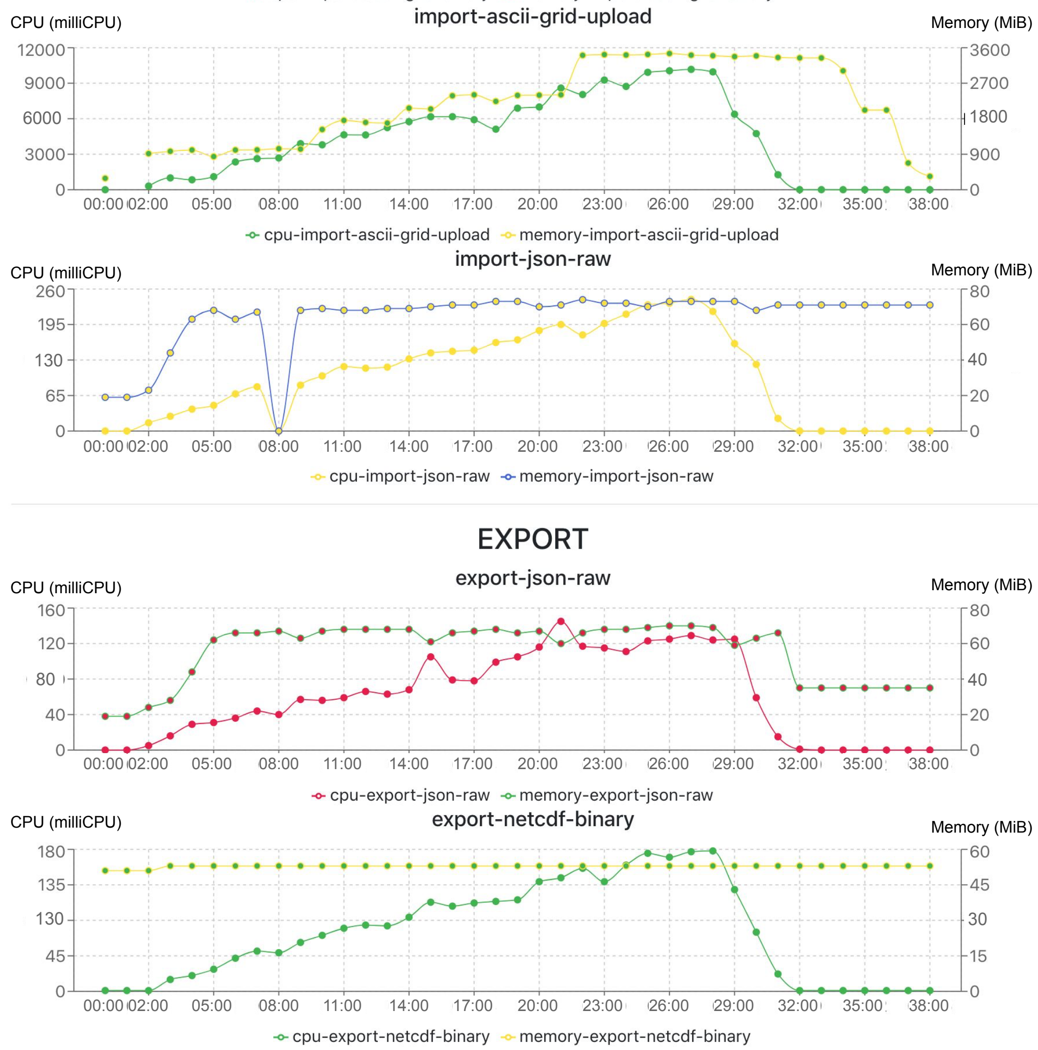

Figure 4.14: Load testing with auto-scaling resource usage of import and export modules over time.

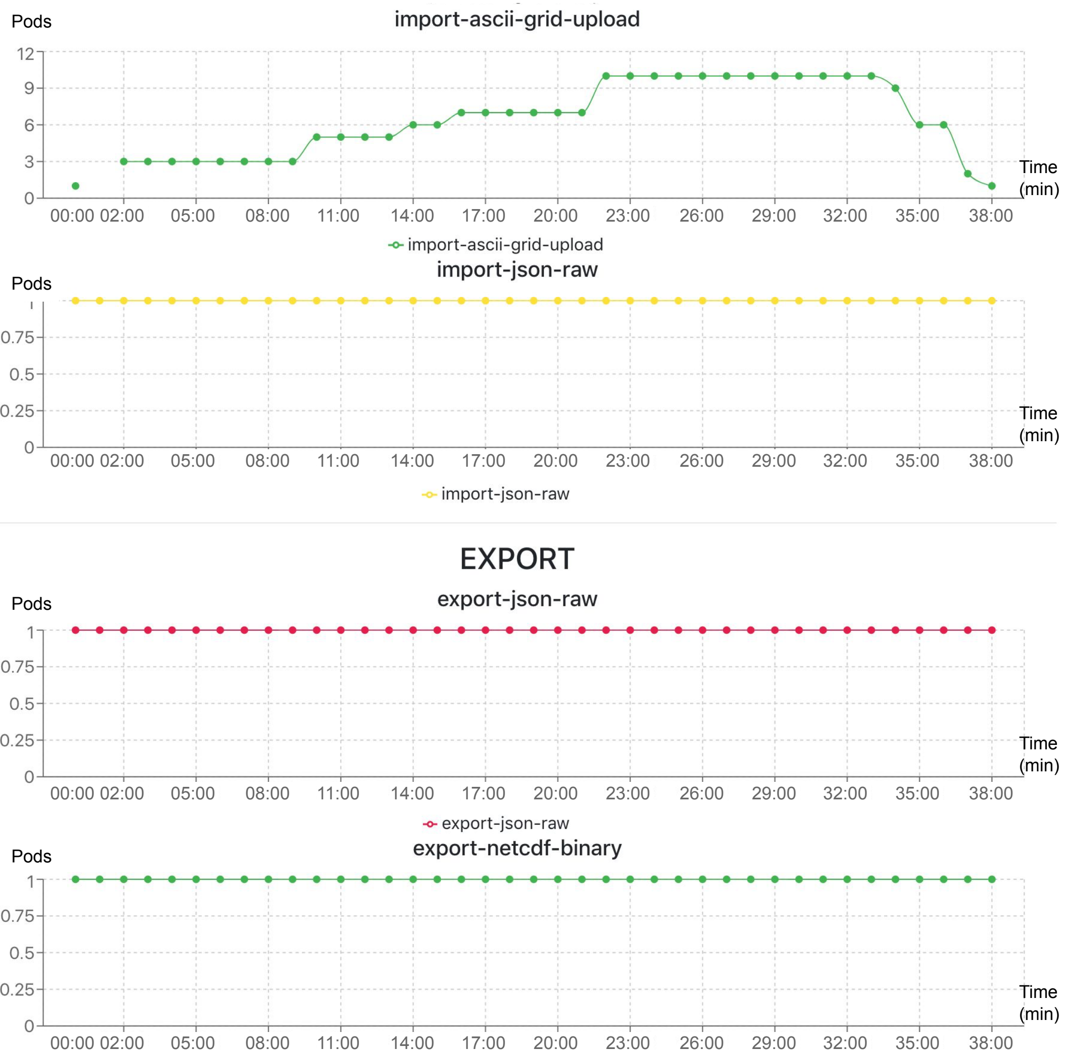

Figure 4.15: Load testing with auto-scaling number of pods of import and export modules over time.

during the test plan. As mentioned in the previous paragraph, we enabled the auto-scaling with a maximum of 10 pods and 80% recommended CPU usage. As the graph shows, we scheduled three pods before initialing the test plan. When the workload increased, K8s spawned new pods while keeping the total CPU utilization up to 80%, as we configured above. After K8s spawned ten maximum pods, it stopped spawning new pods. However, each pod was able to vertical scale up to the limit of 2 CPUs. Even we saw the resources released after the peak load, but the number of pods did not decrease according to the graph. K8s has a threshold of reducing the number of pods, and it waits for 5 minutes before terminating a pod after a resource is released.

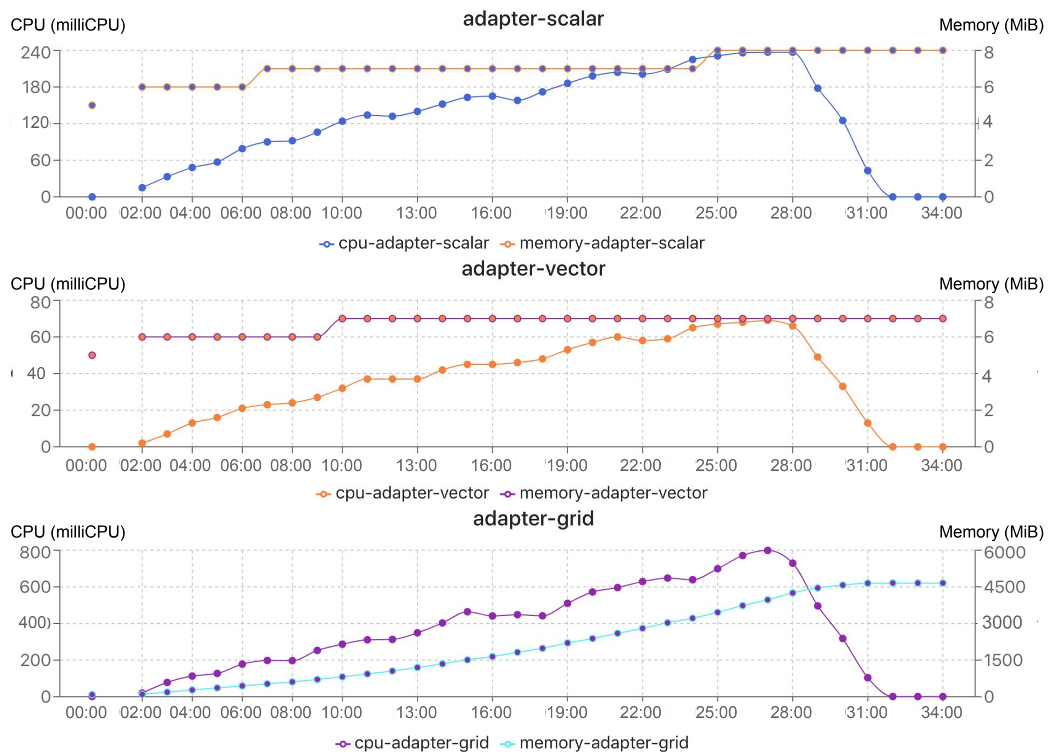

Figure 4.16: Load testing with auto-scaling resource usage of database adapters over time.

Figure 4.16 shows the resource utilization of database adapters in the WDIAS. The scalar adapter and vector adapter are not getting heavy load since they only process a scalar array of data points. However, we can see the CPU usage over time increased according to the test plan workload and cool down after the peak. For the grid adapter, the CPU resource utilization increased and cooled down similarly to other adapters.

However, the memory resources are not released due to the caching of netCDF files by the application.

4.3.5 Query Module Load Test

This Section 4.3.5 discussed the query test plan performance. During the test plan, we performed multiple information retrievals queries such as queries over locations, parameters, and timeseries, which we described in the test planning phase. The test plan performed approximately 123×10^3 of sample requests only with the query microservice during the test period. We performed the query test plan over 5 minutes with a higher number of requests than other test plans with a peak of 600 hits per second.

Table 4.6: Throughput and latency of query test cases with 15-minute data

Label	Samples	Avg	Min	Max	90% Line	Std.Dev.	Error	RPS
* Locations	11462	992	5	17569	1740	704.67	0.00%	38.3
Area Locations	11390	599	2	16755	1047	514.68	0.00%	38.1
Location Parameters	11350	666	1	16821	1179	632.86	0.00%	38.0
Locations Parameters	11305	659	1	17459	1149	678.55	0.00%	37.8
Location Timeseries	11269	647	2	32693	1102	746.51	0.00%	37.7
Locations Timeseries	11237	669	1	32651	1162	889.18	0.00%	37.6
Locations, Parameter Timeseries	11192	662	2	32806	1152	789.49	0.00%	37.4
Area Timeseries	11135	820	2	33235	1461	877.68	0.00%	37.3
Area, Parameter Timeseries	11084	864	1	16699	1584	751.00	0.00%	37.1
*, Parameter Timeseries	11011	696	2	16689	1216	693.33	0.00%	36.8
* Timeseries	10968	1473	24	18283	2467	912.45	0.00%	36.7
TOTAL	123403	794	1	33235	1536	789.76	0.00%	412.6

Table 4.6 shows the response latency summary details. The average latency is much higher than the minimum value. Also, the summary reported higher maximum values as well. Standard deviation showed a quiet higher value, which means the response latency values are not closer to the average value. Noticeably, we cannot see any errors in the table, which means all the requests successfully process. The throughput is similar among all the requests. The performance of the query adapter depends on the performance of the document storage database performance and geo-indexing performance.

Figure 4.17 shows the response latency overtime for the query test plan. When the

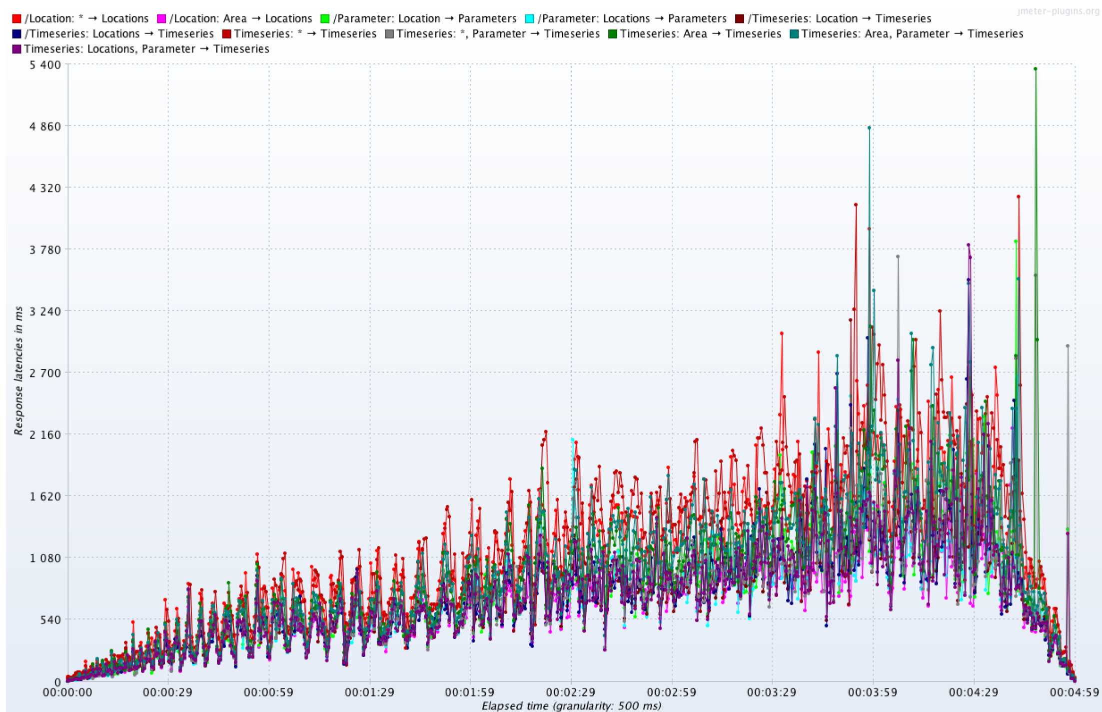

Figure 4.17: Response latency over time while load testing query test over 5 minutes.

number of requests gets higher, the latency also gets increased. Further, Figure 4.18 shows the response latency against the number of active threads for the query test plan. When the number of server hits gets higher, the latency also increases. According to the figures, we can conclude that the performance of the query adapter depends on the database. Even if we increase the number of query adapter pods, the throughput of the microservice is not going to increase until we increase the performance by using the z-axis scaling of scale cube concept.

Whenever the timeseries data is not found in the query adapter, it reads data from metadata adapter and index for search over timeseries metadata. We used the above mechanism to increase the performance with geo timeseries searches. The performance of the document storage can further improve with using its features like replication for high availability, and sharding for higher throughput. We did not attempt those performance improvements during the test plan since it was beyond the scope of the research. However, the users of the WDIAS system can enable such features and get higher performance.

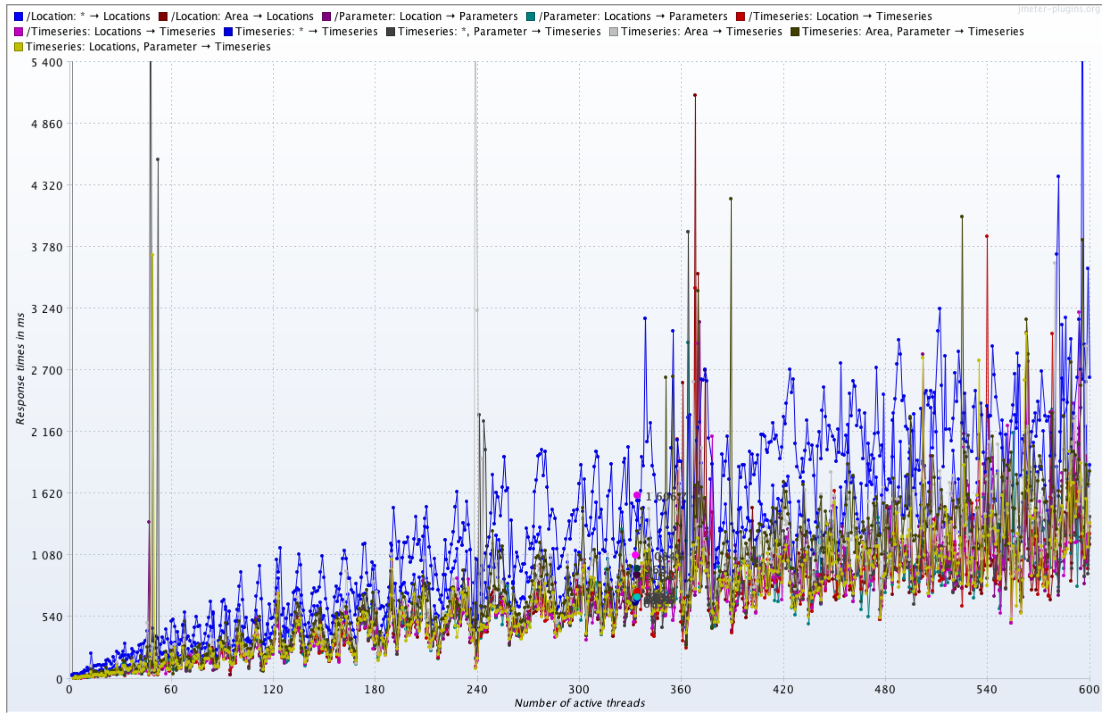

Figure 4.18: Response latency times vs threads load testing query test over 5 minutes.

4.4 WDIAS Performance Evaluation

This section concludes the observations in Section 4.3 against the performance metrics defined in Section 4.1.2.

During each test plan, we showed the latency for data insertion, and retrieval operation type kept constant over the whole test plan run time without any significant change. When we increased the request size from 60-minute resolution data to 15-minute resolution data, the latency also increased throughout the whole test plan with a smaller number. During load testing with 15-minute resolution data with auto pod scaling (see Section 4.3.4) test run, the performance of the grid data got better when compared to load testing with 15-minute resolution data (see Section 4.3.3) without auto-scaling. Thus, we can assume by adding more resources to the WDIAS, it can handle more workload on the system by providing higher throughput.

While keeping the latency constant without significant change, we showed the throughput of the WDIAS kept constant while increasing the request size from 60-minute resolution data (24 data points) to 15-minute resolution data (96 data points)

for all the data types such as scalar, vector, and grid. When the number of active threads increased, the WDIAS able to provide the same throughput while maintaining the latency stable.

One of the reasons for having deviations at the peak time due to uses of a single database instance for the data consistency such as using InfluxDB for scalar and vector data, and netCDF with parallel access support for grid data. Further, it is possible to increase the performance of the before-mentioned bottleneck places such as using the InfluxDB commercial cluster support for high availability or horizontal scaling of InfluxDB. By using the InfluxDB cluster, we can run multiple pods of scalar adapter and vector adapter to support more workload.

Grid adapter is using a Python wrapper for netCDF FORTRAN implementation with parallel IO enabled. However, it has some performance issues, as well as memory leak issues. Since each microservice in the WDIAS is independent of technology, we can use a different technology stack to implement the service. Doing so, we can increase the performance as an example by directly porting the grid adapter to netCDF with a low-level language such as C or FORTRAN. Other than Amazon EKS, users can use a different K8s cluster that supports persistent volumes with the access of Read Write Many [36]. Then they can run multiple pods of grid adapter and increase the throughput of handling grid data.

When we looked into the resource utilization of WDIAS, by using the K8s as the container orchestration system, we were able to scale up and cool down the system as required based on the workload. During the test plan with auto-scaling enabled in Section 4.3.4, we demonstrated the system gets scale up to the maximum at the peak time. Then cool down to a single pod after finishing the test cases. Given the above facts, we can run the WDIAS on a cluster from 1 CPU node to nodes with 100 CPUs. As described in microservice architecture (see Section 3.3), we used many of the concepts of modern microservice architecture to create stateless, failover, redundant microservice to achieve such capabilities.

The WDIAS supports auto-scaling out of the box with K8s support. Services can be configured with a maximum number of pods to avoid over resource usage. When

there is not much workload on the system, the system cools down to fewer pods to save more resources. When there is an issue with a pod, K8s auto-schedule a new pod and remove the unhealthy pod. Also, it allows updating the system without any downtime with rollback updates. By doing that, the K8s create new pods with upgraded microservice and forward the traffic to new pods after spawning enough new pods, and terminate the old pods.

The WDIAS processed many requests with a larger request size than the normal usage with a lower rate of failures to process the requests, mainly with insert grid data. If we want to reduce the risk of unable to process data, then the system can be configured to run with redundant pods to handle spikes of workloads. Also, while configuring the auto-scaling, the K8s can be configured to maintain a lower amount of CPU usage, such as 50% to 60% rather than 80%. Such configuration will always spawn new pods to handle double of current peak load that the system can handle at a given time.

Chapter 5

Summary

This chapter includes the summary of the WDIAS. In Section 5.1, we summarize and conclude the research. Section 5.2 includes the limitations of this research, and future improvements that can increase the performance and usability of the WDIAS.

5.1 Conclusions

To enhance the accuracy of the weather forecast, we need to provide reliable and timely weather data for NWMs. Also, most of the existing weather data integration systems are based on basic client-server architectures or monolithic distributed system architectures or developed based on the best technology available at the time of developing those systems. However, most of the concepts are outdated with the technology advancement, and we cannot get the full advantage of modern computational models such as containerized applications and using the cloud computing to gain benefits of scalability, high availability, and low-cost operations. Even though there are many weather data integration systems, most of them are proprietary or closed source. Thus, for an island like Sri Lanka that has different kinds of weather seasons yearly, those software needs to be highly customize.

In this research, we proposed an extendable open source weather data integration and assimilation system known as Weather Data Integration and Assimilation System (WDIAS) [38]. WDIAS focuses on providing a system to efficiently integrate weather

data from different sources with supporting quality control. Furthermore, the proposed system is compatible with current cloud computing technologies, architecture patterns, and compatible with modern cloud computing architectures.

First, the architecture of WDIAS starts with SOA using an Enterprise Service Bus (ESB) and then moved to actor model-based architecture. After comparing the disadvantages of using such architectures with the system requirements, we came up with modern microservice architecture. One of attractive feature of microservice architecture is independent nature of microservices enables to design a highly scalable modular system which can use to handle large volumes of data using a large resource pool. Also, we used many of the concepts of modern microservice architecture to create stateless, failover, redundant microservice to achieve such capabilities. WDIAS implemented the microservice architecture on top of the K8s, which is a container orchestration platform [39]. We mapped each microservice of WDIAS to a pod in the K8s cluster. Also, each microservice exposed endpoints to the users which are only focus on supporting a specific operation or functionality which follows the concept of smart endpoints dumb pipes. For importing different format data into the system, we added microservice for each data format that known as import microservices. Then, these import microservices store the data via an appropriate database adapter microservice after converting to compatible data format. For handling multiple data formats efficiently, we introduced a database structure in WDIAS. Then, users can fetch data in the required format via using export microservices. Further, we enabled integrating extensional microservices which provides the data preprocessing capabilities, and it allow users to integrate new features as new extension microservices. We used geo-index to support geo-based search queries and metadata search queries. Also, the system provides asynchronous request handling based on the request size with maintaining status per request.

Compared to other existing weather data integration and assimilation systems, WDIAS has the advantage of taking full advantage of cloud computing technologies to implement a highly scalable and available system at low-cost. Because it uses containerized applications to implement the microservices, it allows independent service

deployments. Each microservice can scale independently without affecting other microservices. Further, each microservice could use a suitable technology stack for the task. Delft-FEWS, MADIS, and MSM have platform dependencies, and some of the deployments are performed by copy-pasting source codes. The Delft-FEWS and MSM provide a centralized data storage mechanism that can become a bottleneck while processing a large volume of data in critical situations. MADIS and DIAS also support data storage arrays to store data. WDIAS and LEAD can support a pool of resources based on the requirement. Like other systems, WDIAS provides data preprocessing capabilities, and users can integrate new data preprocessing capabilities by implementing them as independent containerized microservice applications. Even though Delft-FEWS, LEAD, and MSM provide workflow engine support and model execution support, these can be achieved using an external workflow management system by fulfilling the data requirements via WDIAS. Further, WDIAS is an extendable open framework such that similar capabilities can be integrated as a new extension module system. Compared to other existing systems, the LEAD is more scalable and is implemented using SOA architecture. However, with the help of modern container orchestration frameworks, such as Kubernetes and cloud computing technologies, those can provide high scalability and high availability as inherent capabilities. Those are mature enough to provide fault-tolerance automatically [40]. Implementing the deployment units as containerized applications based on the modern architecture patterns allows WDIAS to implement a highly scalable and high available system without heavy effort on those features. A comparison between WDIAS and other systems are presented in Table 3.1.

Based on the microservice architecture patterns and the nature of the weather metadata, we designed database structure of WDIAS to provide higher performance and efficient storage. The system uses a timeseries database to store scalar and vector timeseries data and use netCDF for grid timeseries data. Also, each database has an adapter, and each adapter uses the concept of database per service, which allows each microservice service to scale without getting interference from other microservices. Moreover, it uses the microservice sagas pattern for asynchronous data handling for

requests with a larger size.

Also, the system provides a generic mechanism to integrate new modules as an extension to enhance the features of the system. This capability enables us to integrate weather data preprocessing flows as extension modules. We designed an extension API to easily create and modify the extension triggers at run time without stopping the system or any downtime for the system reconfiguration. The system provides extensive timeseries query endpoints to easy search over the stored timeseries metadata with supporting geo-based queries.

We evaluated the performance using a cloud-based setup on the Amazon Elastic Kubernetes Service (Amazon EKS). Based on the performance analysis, we were able to perform 300 RPS, and the system handled the workload successfully with minimizing the error rate. Also, while we increased the request size from 24 to 96 data points per scalar and vector timeseries, and the number of ASCII Grid files from 24 to 96, WDIAS was able to provide the same throughput while keeping the latency constant. Likewise, the system was able to provide an increase in the throughput by keeping the latency constant while increasing the workload. Thus, we can conclude that the system is scalable. With K8s auto-scaling enabled, the K8s elastically adjust the number of pods according to the workload with provided auto-scaling configurations. The results of WDIAS shows that the system can run on a cluster from few CPU node to nodes with a few hundred CPUs.

5.2 Research Limitations

Next, we discussed *Lack of data preprocessing modules*: We implemented the extension modules as an eco-system of integrating different preprocessing modules as add-ons to WDIAS. Then, users can use these modules for process the data which are inserted into the system in real time or process later based on regular intervals. Also, we provide a more generic open interface approach to create more preprocessing modules and trigger them via the extension API. Since WDIAS is an open-source system, we are expecting that the community will create more modules, and provide them as

contributions for future users. When compared to other systems like Delft-FEWS, the current system only has few extensions for the testing purpose. Thus, we can consider that as one of the major areas that need to be improved.

Grid data performance: As mentioned in Section 4.4, grid adapter is using a Python language wrapper for netCDF FORTRAN implementation with parallel IO enabled. However, it has some performance issues and memory leak issues as well. Since each microservice in WDIAS is independent of technology, we can use a different technology stack to implement the service. As an example, we can use a low-level language such as C or FORTRAN to implement the grid adapter, which will affect performance improvements and lesser issues. There is some future work that can focus on improving the performance of storing grid data.

Improvements to test cases: As described in detail with Section 4.2.1, WDIAS test plans use the time-stepping feedback loop with concurrent threads, which is the best approach support by JMeter at the moment. However, using the above approach, we cannot get the desired RPS for each test case within the test plan. Because of the above issue, WDIAS test plans further improved by removing redundant test cases such as create timeseries at the beginning. Having created timeseries at the beginning causes some issues such as most of the thread are working on that test case, rather than running the required test cases. Thus, the test plans need to be further improved with running more test cases with the desired RPS.

Tune WDIAS database structure performance: As further described in Section 4.4, we can increase the performance of adapter microservices by using InfluxDB commercial cluster support for high availability or horizontal scaling of InfluxDB. With the InfluxDB cluster, it is possible to run multiple pods of scalar adapter and vector adapter to support more server hits per second. WDIAS's performance can be improved by partitioning the timeseries key space into multiple InfluxDB instances. For example, based on the key attribute Section 3.4.1 Timeseries Type, the scalar adapter can connect to four instances of InfluxDB instances such as external historical, external forecast, simulated historical, and simulated forecast.

Explicitly not testing for high availability: By default, the container orchestration

framework handles service failures, auto scaling, auto down scaling, and independent service deployments. Also, when considering about modern cloud resources in data centers have better availability and failure handling. Combining both factors enables high availability for WDIAS-based systems. However, in this research, we do not rely on third party tools. Also, we do not explicitly test for the high availability with performing manually terminating services, and letting crash some of the services to measure how is it affect on WDIAS-based system's high availability.

5.3 Future Work

Here, we discussed the enhancements that we can add to improve the performance and usability of the system.

Supporting irregular grids: When consider about 2D grid, there are two types of grids such as regular grids and irregular grids. In regular grid system, each shell consist of fix width and height. However, for irregular grid system, each shell define as a 2D polygon with there's coordinates. At the moment, WDIAS does not support irregular grids, but it has the API endpoints defined for the implementation of such support with following the microservice architecture. We reduced the scope of the research by removing such components from the system, and such level of implementation does not belong to the scope of WDIAS as well.

Define infrastructure as code: Physical nodes for the K8s cluster can be created using cloud computing providers such as Amazon EKS, Google Cloud, Microsoft Azure cloud, or even using own cluster. During the load testing, we used Amazon cloud resources. Thus, users need to have some understanding of Amazon cloud and its technologies to setup the system, and the K8s cluster. However, the infrastructure that needs to deploy WDIAS can be defined as code such as using tools like Terraform, which is independent of the cloud computing provider. Using such a tool, users will be able to quickly deploy WDIAS on any cloud provider without much hassle. WDIAS resource requirements are configured as code in the Terraform, and user can easily interact with those clouds using Terraform provider modules without much prior knowl-

edge.

Alerting extension modules: At the moment, in the basic framework of WDIAS, the extensions only support OnChange and OnTime event triggers. However, it is useful to have event triggers support based on the configured alert level or threshold level. Here, users should be able to register triggers based on given alert levels, and the alerting extension should trigger a webhook provide by the user while registering the trigger. For example, a user can register an alert event on particular timeseries such as waterlevel, and also provide the threshold value based on historical flood events. When the waterlevel exceeds the given threshold value, the system will trigger the provided webhook. Then the user will be notified or the webhook endpoint will take necessary actions.

Publisher-Subscriber extension modules: WDIAS could be extended to trigger external systems via publisher-subscriber support. For example, we may sync a weather station's precipitation timeseries data into an external system through WDIAS when there is rain. One possibility is, the external system can sync data with timeseries data via polling through WDIAS. However, it is inefficient as the external system needs to fetch the data via export modules, even there is no rain. Instead, we can use publisher-subscriber capabilities to sync the data efficiently. With these extension modules, users can create topics, and publishers extensions can push data into the topics when particular criteria are met, such as when there is rain in the above example. Then external systems could subscribe to the topic above topics while providing a webhook endpoint. Whenever data is pushed into the topic, the extension modules will send data to the subscribers via webhooks.

References

[1] Mesoscale & Microscale Meteorology Laboratory, *Weather research and forecasting model*. [Online]. Available: https://www.mmm.ucar.edu/weather-research-and-forecasting-model.

[2] CUrW SL, *2017 story map in Sri Lanka*. [Online]. Available: https://www.curwsl.org/public/2017-story-map/.

[3] ——, *Observed rainfall of CUrW SL*. [Online]. Available: https://www.curwsl.org/public/observed-rainfall/.

[4] A. Kawasaki, P. Koudelova, K. Tamakawa, A. Kitamoto, E. Ikoma, K. Ikeuchi, R. Shibasaki, M. Kitsuregawa, and T. Koike, "Data integration and analysis system (DIAS) as a platform for data and model integration: Cases in the field of water resources management and disaster risk reduction," *Data Science Journal*, vol. 17, 2018. DOI: 10.5334/dsj-2018-029.

[5] C. H. Macdermaid, R. C. Lipschutz, P. Hildreth, R. A. Ryan, A. B. Stanley, M. F. Barth, and P. A. Miller, "Architecture Of MADIS data processing and distribution at Fsl P2.39," 2005. [Online]. Available: https://pdfs.semanticscholar.org/d315/f09145089bff01d5dc211d6ac7dd78f220d0.pdf.

[6] M. Werner, J. Schellekens, P. Gijsbers, M. van Dijk, O. van den Akker, and K. Heynert, "The Delft-FEWS flow forecasting system," *Environmental Modelling and Software*, vol. 40, pp. 65–77, Feb. 2013. DOI: 10.1016/j.envsoft.2012.07.010.

[7] G. Naveendrakumar, "Five Decadal Trends in Averages and Extremes of Rainfall andTemperature in Sri Lanka," DOI: 10.1155/2018/4217917.

[8] M. G. F. Werner, J. Schellekens, and J. C. J. Kwadijk, "Flood early warning systems for hydrological (Sub) catchments," in *Encyclopedia of Hydrological Sciences*, Chichester, UK: John Wiley & Sons, Ltd, Oct. 2005. DOI: 10.1002/0470848944.hsa022.

[9] C. Haggett, "An integrated approach to flood forecasting and warning in England and Wales," *Water and Environment Journal*, vol. 12, no. 6, pp. 425–432, Dec. 1998. DOI: 10.1111/j.1747-6593.1998.tb00211.x.

[10] T. Kokkonen, A. Jolma, and H. Koivusalo, "Interfacing environmental simulation models and databases using XML," *Environmental Modelling & Software*, vol. 18, no. 5, pp. 463–471, Jun. 2003. DOI: 10.1016/S1364-8152(03)00020-3.

[11] K. Droegemeier, D. Gannon, D. Reed, B. Plale, J. Alameda, T. Baltzer, K. Brewster, R. Clark, B. Domenico, S. Graves, E. Joseph, D. Murray, R. Ramachandran, M. Ramamurthy, L. Ramakrishnan, J. Rushing, D. Weber, R. Wilhelmson, A. Wilson, M. Xue, and S. Yalda, "Service-oriented environments for dynamically interacting with mesoscale weather," *Computing in Science and Engineering*, vol. 7, no. 6, pp. 12–29, Nov. 2005. DOI: 10.1109/MCSE.2005.124. [Online]. Available: http://ieeexplore.ieee.org/document/1524855/.

[12] D. Salas, X. Liang, M. Navarro, Y. Liang, and D. Luna, "An open-data open-model framework for hydrological models' integration, evaluation and application," *Environmental Modelling and Software*, vol. 126, p. 104 622, Apr. 2020, ISSN: 13648152. DOI: 10.1016/j.envsoft.2020.104622.

[13] C. Hewitt, *Why modern systems need a new programming model*. [Online]. Available: https://doc.akka.io/docs/akka/2.5.5/scala/guide/actors-motivation.html.

[14] Akka.io, *When and where to use Akka cluster*. [Online]. Available: https://doc.akka.io/docs/akka/2.5/cluster-usage.html#when-and-where-to-use-akka-cluster.

[15] IBM, *Containerization Explained*. [Online]. Available: https://www.ibm.com/cloud/learn/containerization.

[16] Docker, *App Containerization*. [Online]. Available: https://www.docker.com/resources/what-container.

[17] Christensen Ben, *Don't Build a Distributed Monolith*. [Online]. Available: https://www.microservices.com/talks/dont-build-a-distributed-monolith/.

[18] J. Lewis and M. Fowler, *Microservices*. [Online]. Available: https://martinfowler.com/articles/microservices.html.

[19] ——, *Microservices pattern: Smart endpoints and dumb pipes*. [Online]. Available: https://martinfowler.com/articles/microservices.html#SmartEndpointsAndDumbPipes.

[20] ——, *Microservices pattern: Decentralized data management*. [Online]. Available: https://martinfowler.com/articles/microservices.html#DecentralizedDataManagement.

[21] C. Richardson, *Microservices pattern: Database per service*. [Online]. Available: https://microservices.io/patterns/data/database-per-service.html.

[22] ——, *Microservices pattern: Sagas*. [Online]. Available: https://microservices.io/patterns/data/saga.html.

[23] Redis, *Redis documentation*. [Online]. Available: https://redis.io/.

[24] InfluxDB, *InfluxDB documentation*. [Online]. Available: https://docs.influxdata.com/influxdb/.

[25] Unidata, *Network Common Data Form*. [Online]. Available: https://www.unidata.ucar.edu/software/netcdf/.

[26] MongoDB, *MongoDB geospatial queries manual*. [Online]. Available: https://docs.mongodb.com/manual/geospatial-queries/.

[27] Google, *Google public dataset - Countries*. [Online]. Available: https://developers.google.com/public-data/docs/canonical/countries_csv.

[28] Apache Software Foundation, *Apache JMeter*. [Online]. Available: https://jmeter.apache.org/.

[29] H. M. G. C. Karunarathne, *WDIAS distributed performance testing based on JMeter*. [Online]. Available: https://github.com/wdias/wdias-performance-test.

[30] ——, *WDIAS performance test plan*. [Online]. Available: https://github.com/wdias/wdias-performance-test/blob/master/test-plan/TEST_PLAN.md.

[31] ——, *WDIAS setup on Amazon elastic Kubernetes service*. [Online]. Available: https://github.com/wdias/wdias/blob/master/docs/Amazon_EKS.md.

[32] Linux Foundation, *Kubernetes: Production-grade container orchestration*. [Online]. Available: https://kubernetes.io/.

[33] CNCF, *Helm documentations*. [Online]. Available: https://helm.sh/.

[34] H. M. G. C. Karunarathne, *Helm charts for WDIAS deployments*. [Online]. Available: https://github.com/wdias/wdias-helm-charts.

[35] Apache Software Foundation, *JMeter: Transaction throughput vs threads plugin*. [Online]. Available: https://jmeter-plugins.org/wiki/TransactionThroughputVsThreads/.

[36] Linux Foundation, *Kubernetes persistent volumes*. [Online]. Available: https://kubernetes.io/docs/concepts/storage/persistent-volumes/#access-modes.

[37] ——, *Kubernetes: Managing compute resources for containers*. [Online]. Available: https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/.

[38] H. M. G. C. Karunarathne and H. M. N. Bandara, *Weather data integration and assimilation system (WDIAS) source code*. [Online]. Available: https://github.com/wdias/wdias.

[39] H. M. Gihan Chanuka Karunarathne, H. M. Dilum Bandara, and S. Herath, "WDIAS: A Microservices-Based Weather Data Integration and Assimilation System," in *MERCon 2020 - 6th International Multidisciplinary Moratuwa Engineering Research Conference, Proceedings*, Institute of Electrical and Electronics Engineers Inc., Jul. 2020, pp. 289–294, ISBN: 9781728190594. DOI: 10.1109/MERCon50084.2020.9185270.

[40] R. Scolati, I. Fronza, N. El Ioini, A. Samir, and C. Pahl, "A containerized big data streaming architecture for edge cloud computing on clustered single-board devices," in *CLOSER 2019 - Proceedings of the 9th International Conference on Cloud Computing and Services Science*, SciTePress, 2019, pp. 68–80, ISBN: 9789897583650. DOI: 10.5220/0007695000680080.

Appendix A

WDIAS REST API

Timeseries Metadata endpoints allow to register new timeseries into the WDIAS system before upload data.

- parameter
  - `GET /parameter` – Get parameters
  - `POST /parameter` – Create parameter
  - `PUT /parameter/id` – Update parameter
  - `DELETE /parameter/id` – Delete parameter
- timeStep
  - `GET /timestep` – Get TimeSteps
  - `POST /timestep` – Create TimeStep
  - `PUT /timestep/id` – Update TimeStep
  - `DELETE /timestep/id` – Delete TimeStep
- location
  - `GET /location` – Get location points
  - `POST /location` – Create location point
  - `PUT /location/id` – Update location point

  - `DELETE /location/id` – Delete location point
  - `GET /location/regular-grid` – Get regular grids
  - `POST /location/regular-grid` – Create regular grid
  - `PUT /location/regular-grid/id` – Update regular grid
  - `DELETE /location/regular-grid/id` – Delete regular grid

- timeseries
  - `GET /timeseries` – Get timeseries
  - `POST /timeseries` – Create timeseries
  - `PUT /timeseries/id` – Update timeseries
  - `DELETE /timeseries/id` – Delete timeseries

Import endpoints allow to upload data into the WDIAS system.

- `POST /import/json/raw` – Insert Scalar/Vector timeseries data with JSON body
- `POST /import/csv/raw` – Insert Scalar/Vector timeseries data in CSV format
- `POST /import/ascii-grid/upload` – Upload set of ASCII Grid files
- `POST /import/ascii-grid/binary` – Upload a single ASCII Grid file

Export endpoints allow to get/download data from the WDIAS system.

- `GET /export/json/raw` – Get Scalar/Vector timeseries data in JSON format
- `GET /export/json/raw?start=2000-01-01&end=2000-01-03` – Get Scalar/Vector timeseries data in JSON format for given time period
- `GET /export/netcdf/binary` – Download Grid data timeseries data in netCDF format

Appendix B

WDIAS Query API

Timeseries Metadata Queries

Get timeseries metadata by timeseriesID.

```
1    GET <HOST_ADDR>/metadata/timeseries/<TIMESERIES_ID>
```

Search for timeseries metadata with key attributes of the timeseries.

```
1    GET <HOST_ADDR>/metadata/timeseries?<QUERY_STRING}
```

QUERY_STRING can be consist of zero to multiple parameters. Multiple query parameters are separated by the ampersand, ”&”.

- `moduleId` (Optional), e.g., HEC-HMS
- `valueType` (Optional), e.g., Scalar
- `parameterId` (Optional), e.g., O.Precipitation
- `locationId` (Optional), e.g., wdias_hanwella
- `timeseriesType` (Optional), e.g., ExternalHistorical
- `timeStepId` (Optional), e.g., each_15_min

For the moment, query are only support by strict matching with above parameter values.

Timeseries Geo-queries

Query Locations within Area – Following endpoint supports to query the locations which exist within the area provided by the GeoJson data in the JSON payload.

```
1    POST <HOST_ADDR>/query/location
2    JSON Body:
3    {
4        "geoJson": {
5            "type": "Polygon",
6            "coordinates": [
7                [
8                    [
9                        <longitude>,
10                       <latitude>
11                   ],
12                   ...
13               ]
14           ]
15       }
16   }
```

paragraphQuery parameters in Locations– Following endpoint supports to query the parameters stored against given locations. Location IDs should be provided as a list of JSON payload.payload.

```
1    POST <HOST_ADDR>/query/parameter
2    JSON Body:
3    ["<locationID>", ...]
```

Query Timeseries by Location – Following endpoint supports to query the time-series metadata for a given location.

```
1    POST <HOST_ADDR>/query/timeseries
2    JSON Body:
3    {
4        "location": "<LOCATION_ID>"
5    }
```

Query Timeseries by Locations

Following endpoint supports to query the timeseries metadata for a given location set. Location IDs should be provided as a list in the locations field.

```
POST <HOST_ADDR>/query/timeseries
JSON Body:
{
    "locations": ["LOCATION_ID", ...]
}
```

Query Timeseries by Parameter

Following endpoint supports to query the timeseries metadata by parameter for given location/locations.

```
POST <HOST_ADDR>/query/timeseries
JSON Body:
{
    "locations": [""],
    "parameter": ""
}
```

Query Timeseries in Area – Following endpoint supports to query the timeseries metadata within a given area (specified as a polygon in geoJson format).

```
POST <HOST_ADDR>/query/timeseries
JSON Body:
{
    "geoJson": {
        "type": "Polygon",
        "coordinates": [
            [
                [
                    <longitude>,
                    <latitude>
                ],
                ...
            ]
        ]
    }
}
```

Query Timeseries in Area by Parameter – Following endpoint supports to query the timeseries metadata within a given area for a given parameter. This endpoint is an extension of the above endpoint.

```
POST <HOST_ADDR>/query/timeseries
JSON Body:
{
    "geoJson": {
        "type": "Polygon",
        "coordinates": [
            [
                [
                    <longitude>,
                    <latitude>
                ],
                ...
            ]
        ]
    },
    "parameter": "<PARAMETER_ID>"
}
```

Query All Timeseries – Send a request with an empty JSON body that will respond with all the timeseries.

```
POST <HOST_ADDR>/query/timeseries
JSON Body:
{}
```