DESIGNING TIME SERIES DATA STORAGE SYSTEMS THAT BALANCE PERFORMANCE, USABILITY, AND MULTI-TENANCY

By

GARY MICHAEL FITZGERALD II

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE

UNIVERSITY OF VIRGINIA
School of Engineering and Applied Science

MAY 2022

© Copyright by GARY MICHAEL FITZGERALD II, 2022
All Rights Reserved

© Copyright by GARY MICHAEL FITZGERALD II, 2022
All Rights Reserved

To the Faculty of the University of Virginia:

The members of the Committee appointed to examine the thesis of GARY MICHAEL FITZGERALD II find it satisfactory and recommend that it be accepted.

Benton Calhoun, Ph.D., Chair

Brad Campbell, Ph.D.

Daniel Graham, Ph.D.

Neal Magee, Ph.D.

ACKNOWLEDGMENT

We thank Victor Ariel and Nabeel Nasir, Ph.D. candidates in the computer science department at the University of Virginia, for their continued help and support in debugging the system. Their devices and time series data needs helped evolve the idea of the SIF platform into a simpler, more stable, and overall more usable technology stack. Additionally, although our final implementation of the SIF platform did not make use of CCRi's services, we also extend thanks to them for the contributions they made during our time working with Optix.earth. Our interactions with CCRi and Optix.earth were critical in forming a mental model for what we wanted in a time series data management platform.

DESIGNING TIME SERIES DATA STORAGE SYSTEMS

THAT BALANCE PERFORMANCE, USABILITY,

AND MULTI-TENANCY

Abstract

by Gary Michael Fitzgerald II,
University of Virginia
May 2022

The number of Internet of Things, or IoT, devices is growing rapidly (a 214% increase from 2015 to 2020). Many expect the trend to continue, with some forecasts predicting 140% growth through 2025. An area which stands to benefit from this expansion is the deployment of time-series data-collecting devices, however, current methods of storing and interacting with time series data are typically limited in one of two ways. Some systems are user-friendly at the cost of query performance or maximum ingest rate. Others have great performance but a more difficult user experience. To get around these issues, we propose a new method for designing time series data management systems. Our method takes isolated features that users desire, like flexible data ingestion and automated database configuration, and offers guidance for how to combine them with a simple and scalable back-end architecture. Using the new method, we built the Smart Infrastructure Foundation (SIF), a system optimized for the collection and storage of sensor data from University buildings, which improves upon existing solutions by offering a greatly simplified user experience without sacrificing generality, scalability, or efficiency.

TABLE OF CONTENTS

Page

ACKNOWLEDGMENT . . . iii

ABSTRACT . . . iv

LIST OF TABLES . . . vii

LIST OF FIGURES . . . viii

1 Introduction . . . 1
- 1.1 Thesis Statement . . . 3
- 1.2 Contributions . . . 3

2 Literature Review . . . 5

3 Current Technology . . . 8
- 3.1 Amazon Timestream . . . 8
- 3.2 Optix.earth . . . 11

4 Conceptual Framework . . . 15
- 4.1 A new time series data model . . . 15
  - 4.1.1 Model vocabulary . . . 16
  - 4.1.2 Model data format . . . 18
- 4.2 Decoupling from a database . . . 19
- 4.3 System-wide scalability . . . 21
- 4.4 Zero-configuration data ingesting . . . 22

5 SIF Design . . . 24
- 5.1 Architectural overview . . . 24
- 5.2 Underlying database technology . . . 25
- 5.3 Ingest component . . . 27

5.4 Insertion component . . . 33
5.5 API . . . 35
5.6 Website . . . 38
5.7 Data visualization tools . . . 38

6 Evaluation . . . 40
6.1 Dense and historical data . . . 40
6.2 Real-time, diverse, and metadata-rich data . . . 41
6.3 A custom data source . . . 42
6.4 A survey of the system in the context of our goals . . . 42

7 Future work . . . 44
7.1 Metadata malleability . . . 44
7.2 Security improvements . . . 45
7.3 MQTT broker software . . . 45

8 Conclusion . . . 47

REFERENCES . . . 50

APPENDIX

A SIF User's Handbook . . . 52

LIST OF TABLES

5.1 An example of what raw data collected from the kitchen example might look like before compression occurs. 26

5.2 A compressed version of Table 5.1. The ingest component automatically configures TimescaleDB's compression algorithm on a per-app basis. 27

5.3 The definitive list of errors that can be reported by the ingest and insertion components and their respective meanings. 35

LIST OF FIGURES

3.1 An overview of the Timestream platform architecture, highlighting its three-layer design [17]. 9

3.2 A simplified architecture diagram of the Optix.time system when deployed in AWS. Certain details not relevant to our discussion are excluded. 11

5.1 An architectural overview of the SIF platform 25

5.2 Internal design of the ingest component. Items above the dotted line are external to the ingest component and are included for clarity. 28

5.3 After a user has logged into the SIF website, they are greeted with a landing page which provides a list of their current apps. 37

Chapter One

Introduction

In recent years, the computing industry has played witness to incredible growth in the Internet of Things (IoT) sector. Forecasters do not see this trend subsiding any time soon, making IoT devices and their surrounding infrastructure a market area with ample room for expansion and innovation [14]. As the IoT expands, the ubiquity of deployed sensors (e.g. temperature, humidity, and air quality sensors placed throughout buildings) and volume of data collected also stand to increase. The data typically collected by such devices is stored as a sequence of correlated timestamps and measurements known as time series data. Many solutions to storing this general form of data have been proposed and implemented in industry and academia to varying degrees of success [5]. These solutions vary greatly and range from complete systems designed to handle data ingest, storage, and retrieval, such as Optix.earth, to strictly database solutions optimized for the sorting, storing, and retrieval of time series data, like Amazon's Timestream cloud service [3, 17]. Generally speaking, these ready-made solutions offer their users the ability to upload time series data, store it in some efficient way, retrieve and/or process the data at a later date, and may offer additional features designed to make time series data more useful or easier to work with, such as measurement metadata.

These existing solutions for managing time series data can easily come up short when considering many practical, real-world use cases. For instance, our research group needed a single unified system owned and managed by a single authority so we could avoid spending

time setting up a new time series database every time one was needed. This alone is not enough to warrant a problem, but the central ownership model became an issue when we also wanted to allow everyone to use this centralized service simultaneously without interfering with one another or requiring inter-user coordination. In particular, the class of prebuilt solutions tends to be lacking with regards to this multi-tenant case, varying data formats, and enabling access to users who are more interested in analyzing data rather than of the specifics of how the data must be represented in transit or a database. More custom solutions can be created by starting with some existing database technology, like InfluxDB, and adding the necessary infrastructural layers to make a more comprehensive data management platform. Although custom solutions can solve some of these problems, they, like many of the prebuilt solutions, tend to result in a system which is lacking in one of two ways: first, some systems prefer raw database performance over usability, making it difficult, confusing, or time consuming to get data in or out of the system. Second, system designers can come at the problem from the opposite direction and value usability over back-end performance.

Continuing to manage our time series data with the existing industry tools or half-measure custom solutions will lead to several issues as the IoT grows. With the tools available now, users will have to deal with data format issues or build their own ingestion pipeline to handle formatting concerns. If multiple tenants want to exist within the same system, they will either need to coordinate to ensure they do not accidentally mix data, or develop mechanisms to mitigate this concern altogether. If no multi-tenant controls are desired, users will need to manually coordinate with one another to ensure different data series are not combined. In summary, users of these systems will be forced to make a decision between taking on more configuration tasks or investing the effort to design a custom solution to meet their needs. Both of these consequences are laborious and ultimately distract from the goal of the system: collect, store, and retrieve time series data.

To circumvent these issues, we first propose a new model for how to think about time series data. In this model, we define generic vocabulary for describing time series data as

well as a high level data format. Our new model allows us to think about such data in an abstract way and reason about how management platforms ought to operate. Using the model, we then propose a set of fundamental design principles for modern time series data management platforms as we step into a future where there are more people using more devices to log more time-based information. Among these principles are scalability, configuration-less data ingestion, and avoiding reliance on a particular database technology. Within the confines of our proposed model for time series data and while following our set of design principles, we implement the Smart Infrastructure Foundation (SIF), a cloud-based time series data management platform. We explore how SIF compares to existing platforms and to the custom storage solution currently in place at the University of Virginia's Link Lab, and conclude with a discussion of how SIF could be improved in the future based on the lessons we learned throughout this process.

1.1 Thesis Statement

Existing time series data management solutions fail to address a multi-tenant, general purpose use case and typically do not balance back-end performance with front-end usability. A new time series data platform that embraces the design principles of using an abstract data model, remaining decoupled from a database implementation, promoting scalability, and requiring zero configuration for data ingestion is able to maintain high performance while also being usable across a diverse set of application use cases.

1.2 Contributions

The work we present advances the ways in which we think about designing complete time series data management platforms. Specifically, we highlight the following key contributions:

- We define three design principles and a new data model intended to help system ar-

chitects avoid issues typically associated with existing time series data management platforms, particularly issues with multi-user operation.

- We develop key system abstractions that enable the cloud-based system to balance performance and usability.

- We create an implementation of a cloud-based time series data management platform built in accordance with our design principles, provided as an aid for understanding how the design principles impact system development.

Funding for this project was provided by the University of Virginia's Strategic Investment Fund [20].

Chapter Two

Literature Review

There have been many academic efforts to improve time series data management systems, ranging from new database optimization techniques to explorations of enhanced IoT device integration. Generally speaking, current efforts are primarily focused on individual components – databases, in particular – rather than entire systems, which is where our work is based.

Deri, et al. describe a new approach for a back-end database system optimized for the storage of high-volume time series data sets [4]. Specifically, they offer reasons as to why traditional relational databases are, on their own, not well suited to handle time series data, including the fact that large relational tables may have indexes too large for the cache and a lack of time-series-friendly data compression. To remedy these problems the authors propose "tsdb", a new database system aimed at making a more performant back-end service for storing time series data. While this new approach makes compression of time series data more efficient and query-able, it does not address usability concerns and is also now positioned in a market occupied by numerous open source and freely available time series database services.

Other authors have made similar efforts focused specifically on the database portion of a time series data management system. Yang, et al. propose a novel time series database technology, EdgeDB, designed to run on edge devices, rather than on some central or centrally

distributed system [22]. EdgeDB's relaxed resource requirements make it ideal for edge devices and they claim the full potential of an IoT cloud can be unlocked by distributing database work across many of these devices. However, this model assumes the presence of edge devices as part of the IoT and data management platform architecture, which we do not believe will be realistic in all use cases. We proceed in our work without the assumption of compute-capable edge devices.

Wang, et al. also propose a back-end database which focuses on operating on edge devices [21]. Unlike existing technologies like OpenTSDB, KairosDB, and InfluxDB, Apache IoTDB has both substantially improved compression efficiency for time series data (compared to OpenTSDB and KairosDB) as well as stronger query performance when aggregating data over long time periods (compared to InfluxDB). While Apache IoTDB offers greatly improved database performance, their edge-to-cloud demonstration applies only to a relatively narrow use case which assumes the presence of an edge device running a database instance. We believe this use case can be generalized to broaden usability while retaining some baseline standard of back-end database performance.

Pelkonen, et al. approach the database problem from the perspective of query performance. They propose a new time series database, Gorilla, which operates on primarily in-memory storage and improves upon existing databases by exploiting the extra speed afforded by avoiding disk interactions [8]. Gorilla essentially acts as a "write-through" cache, holding twenty-six hours worth of data in memory and sending the rest to HBase for long term storage. As a purely back-end technology, Gorilla does little to solve the issues we see with time series data before the database is even reached, such as various data formatting problems. We believe a more well-rounded approach including components outside of the database is necessary to create a system which more appropriately balances the concerns of usability and back-end performance.

Rhea, et al. highlight Cisco Meraki's usage of the LittleTable relational time series database since 2008. LittleTable is highly performant, storage efficient, distributable, and

fault tolerant [10]. However, in their presentation of the LittleTable system, there is no mention of a metadata management system. This missing feature makes the organization of data stored in the system of critical importance. Although this organization is possible internally at Cisco, it is unlikely that such organization would persist in a similar system which was open to the public.

Meehan, et al. recognize the challenges of scaling IoT-connected database systems using existing architecture which follows the "extract, transform, and load" data ingestion model [6]. They claim this model is cumbersome and tends towards implementations which are slow and will ultimately be unable to cope with the rapidly growing IoT. Meehan, et al. propose a new ingestion model which uses an in-data-stream architecture to apply the necessary transformations and cleaning to incoming data as it arrives. This stream-based architecture bears some resemblance to the platform architecture we develop later in Chapter 5, however does not make any attempt to improve the usability aspects beyond that of existing tools.

Continuing with an exploration of the time series data ingestion process, Arman, et al. propose automated mechanisms for data ingestion and regularization [2]. While this work provides ample improvement over prior art in terms of usability by making data ingestion easier, it does not address the back-end efficiency associated with their mechanisms or address database concerns. Additionally, the scalability of their technique is not fully explained.

Chapter Three

Current Technology

Before taking our own approach, we looked at several existing solutions to the problem of collecting, storing, and managing time series data. In this chapter we will outline two such solutions and discuss their basic architectures, benefits, drawbacks, as well as share any special insights learned from our time working with them.

3.1 Amazon Timestream

Timestream is Amazon's solution to the problem of storing and retrieving time series data [1]. They leverage their existing cloud infrastructure to create a cloud-based database service optimized specifically for data organized by timestamps, then add additional layers for data ingestion and querying [17]. The resulting three-layer system is efficient and performant, however, as we will demonstrate, it makes many of the mistakes we described earlier, including a lack of consideration regarding how it interfaces with users, manual configuration ahead of data ingestion, and high costs when ingesting small data packets.

Timestream's architecture, shown in Figure 3.1, is fairly straightforward. The top layer of the system, the data ingestion layer, allows users to send data to the system by invoking certain methods from the AWS SDK. After it has been uploaded to the system, the data is processed and prepared to be placed into the storage layer, where Timestream is really able to

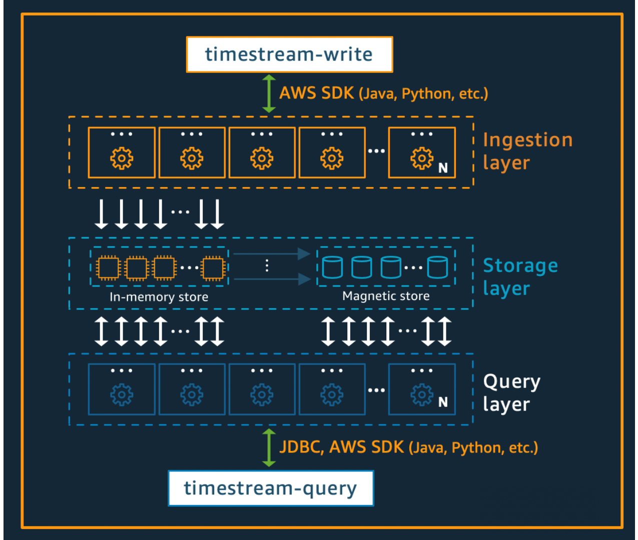


Figure 3.1 An overview of the Timestream platform architecture, highlighting its three-layer design [17].

shine. In this storage layer, Amazon makes use of several interesting storage mechanisms to improve query speed and storage costs. Their insight is to automatically move data between the more expensive solid state, or in-memory, storage and cheaper magnetic storage based on the likelihood of that data being queried for. Using the assumption that more recent data is more likely to be queried for and, therefore, should have less query latency, Timestream records the most recent data points in higher-speed, more costly solid state storage. After a period of time has passed, the older data is asynchronously transferred to less expensive, slower magnetic storage and deleted from the solid state storage pool. Despite this ongoing manipulation of the storage location, the system's user-interface (ingest and query APIs) present a unified front that hides this optimization from the user.

Timestream offers some additional benefits, but a quick overview of their storage-layer terminology is required to fully appreciate them. Similarly to the system we propose below in Chapter 4, Timestream operates on a model for how Amazon envisions time series data. In this model, the system's storage layer contains one or more databases. Each database contains at least one table, and each table is host to sets of related time series data. For instance, one table could contain weather-related data sets like temperature, humidity, and

air quality, while a second table contains hardware usage data sets, like CPU and memory usage, hardware temperatures, and so forth. Since a database isolates groups of tables, it could be used to represent a collection of tables from a single project, person, or organization. A set of time series data in Timestream is composed of a collection of records, which can be thought of as rows in a SQL database. Each record contains a timestamp and at least one measurement or value. On a per-record basis, additional metadata can be provided to describe how the measurement was collected or other relevant information concerning that record. These metadata key/value pairs are referred to as dimensions, where each dimension has a name and value [18]. For instance, a set of data points measuring temperature over time in a building could have a dimension called "location" and a value corresponding to where the data was collected, such as "office". Using this storage-layer terminology, we can discuss some additional benefits of using Timestream. Firstly, Amazon's system supports multi-measure records, allowing for a single row to contain multiple measurements. As an example, a server computer could report CPU, memory, and network usage simultaneously and Timestream can record them all in the same record, reducing the space required to store the measurements. Second, the system supports unlimited dimensions and dimension values, a feature that is not universally true of time series data management platforms and will appear again in our discussion of Optix.earth, below.

Despite these benefits, Timestream does fall short of an ideal time series data management system. First, it is not friendly towards small write requests. Since the service is billed based on the number and size of writes, one might initially think that a reasonable number of very small writes would be economical. However, since the minimum billed write size is 1KB, smaller write requests end up being financially wasteful. Amazon recommends users of their service add a batching layer which collects write requests and dumps them into Timestream in larger, bulk requests [19]. This batching layer creates a number of issues, such as having to build out an entire fourth layer of infrastructure with an efficient and reliable implementation. Furthermore, Timestream has the same issues as many other existing platforms when it comes

to supporting multiple tenants, users who don't want to deal with setting up the AWS SDK, and users who need to deploy many devices and lack the time to repeat the configuration steps for each device. A final point of issue with the system is that it fails to provide read consistency across the database. Specifically, certain write requests can take up to six hours to fully process, depending on where Timestream decides to record them (in solid state or magnetic storage). Batches of write requests which are followed immediately by queries are by no means guaranteed to be reflected in the results of those queries. As proposed, our platform implementation is more cost effective for small database writes without the need for additional infrastructure, supports an unlimited number of simultaneous tenants with no concerns of users depositing data into the shared storage locations, avoids the long-duration delays between sending in a write request and the measurement appearing in the database, and maintains scalability in the process.

3.2 Optix.earth

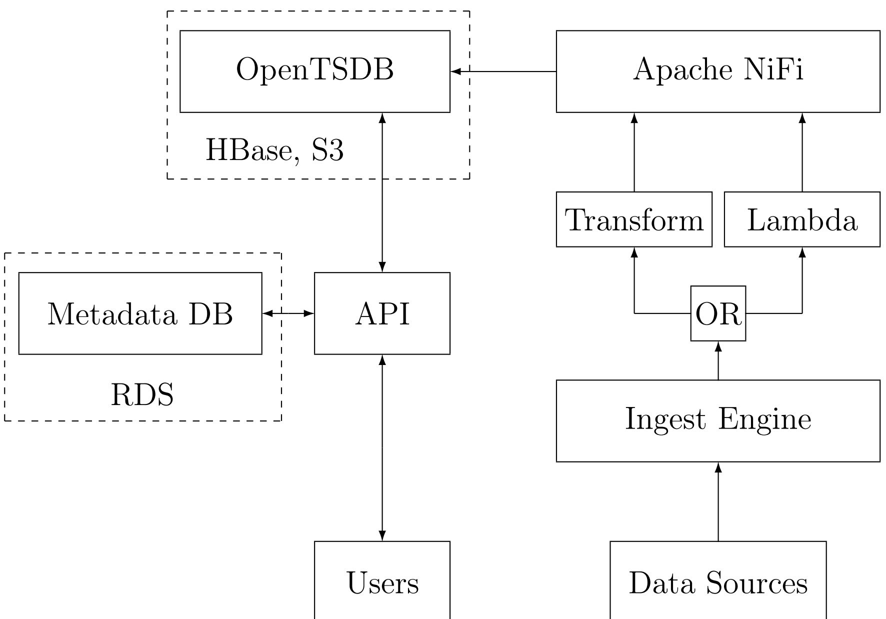


Figure 3.2 A simplified architecture diagram of the Optix.time system when deployed in AWS. Certain details not relevant to our discussion are excluded.

Optix.earth (Optix) is an end-to-end, closed-source time series data storage and querying solution offered by General Atomics Commonwealth Computer Research, Inc. (CCRi) via a premium subscription [3]. Optix's technology stack, depicted in Figure 3.2, includes a Java ingest engine, fully customizable data transformation modules, automatically scalable storage via HBase, and utilizes the open-source program OpenTSDB as the underlying data storage mechanism. Optix was, at the outset of our work, intended as a replacement for our lab's existing time series ingestion and storage pipeline due to a number of benefits it appeared to have. After a deep exploration of the Optix system, we determined that it and our time series database needs were not compatible. We also see room for improvement with regards to supporting more general purpose use.

As a stand-alone product, Optix's arguably best feature is that it completely abstracts the database from the rest of the system. Instead of thinking about storing information in database terms, the system offers a storage API consisting of data submission and retrieval endpoints and enables users to think about their data in simpler terms. On the data ingestion side, Optix offers a highly customizable and flexible ingest pipeline consisting of a Java ingest engine and optional support for user-written Amazon Lambda functions to be used as data transformers. These custom transformation functions make it incredibly easy to get new devices sending data into the system and allow the user to edit their transformation functions at any time in response to changing circumstances. Unfortunately, though, these benefits come at the expense of scalability and user experience.

Optix's Java ingest engine offers excellent ingest performance when using data transformation functions written in Java and compiled at run-time (somewhere in the millions of data points per second, according to CCRi). However, these compiled transformation functions require manual recompilation when modified. This is fine if there are not many users or transformation functions, but as the system scales up, it becomes a problem for those administering the system because this process is difficult to automate effectively and safely. Considering the issues with this recompilation step, Optix users tend towards using Lambda

functions as their method-of-choice for implementing data transformation logic. During our time using the system, we essentially used Lambda functions as the exclusive method of data transformation because of their simplicity and the ability to modify them on-the-fly. After extensive and comprehensive experimentation with Optix, including getting devices to stream data to the system, building a web front-end on top of their API, etc, we began to encounter significant problems as we tried to scale up the system. With a higher ingest rate (around twenty data points per second instead of one or two), Optix began to exhibit ingest lag. Similarly to the phenomenon we described when discussing Timestream, data points did not land in the Optix database until hours after they had been uploaded. We later discovered through working with CCRi that this issue is due to latency originating from the invocation of Lambda functions in AWS, making this feature of the Java ingest engine essentially useless beyond data sources with very low input rates. With the only option now being to write the data transformation functions in Java, it became clear Optix requires additional layers of infrastructure in order to broaden its usability.

These additional layers are where Optix begins to exhibit some of the same flaws as many other platforms. In an effort to appeal to users by offering an ingest pipeline that is easy to set up and customize, Optix lost significantly on performance. In doing so, the system falls victim to the database-performance-versus-system-usability trade-off. However, even if this system was able to solve the ingest performance problem with Lambda functions by switching to purely Java-based transformation functions, it makes the same mistakes as Timestream did by failing to support basic requirements that many platforms will encounter, like supporting multiple tenants while avoiding the mixing of data. Also, without the addition of supplemental infrastructure layers, Optix violates the database decoupling and zero-configuration data-ingest principles we will discuss in the next chapter.

In spite of the many issues we discovered while exploring the Optix system, we believe it helps demonstrate the importance of our proposed design methodology. Instead of focusing primarily or exclusively on database or user interaction concerns, designers of time series data

management systems need to consider each and ensure their platform effectively compromises when there are differing goals or is able to build bridges to ensure both sides get what they need, like when users would like to submit many data formats but the database can only understand one format. It also validates our decoupled database principle, described below, because a system which inherits flaws from its underlying database is, itself, a flawed system. Optix shows us that an ideal platform will not expose database nuances, like the inability to store unlimited metadata values, to its users.

Chapter Four

Conceptual Framework

In this chapter, we will discuss the key insights and fundamental principles used to shape the design of the SIF platform. These include the creation of an abstract model for time series data sets, the decoupling of this model from any specific underlying database technology, system-wide scalability, and zero-configuration data ingesting. For each principle, we will explain precisely what it means in the context of time series data and why we believe it is of critical importance.

4.1 A new time series data model

To create a system that effectively and efficiently handles time series data, we need to define what exactly this type of data is, what it looks like, and how it can be characterized. Consequently, the first and foremost design principle we considered was the creation of an abstract time series data model. Our model consists of two parts: vocabulary describing the components of time series data and a standardized data format to describe the individual data points of a series.

4.1.1 Model vocabulary

To aid with the definition and explanation of our model's vocabulary, we will reference a simple scenario where a researcher has deployed four sensors: two in a kitchen and two in an office. In each room, one of the sensors measures temperature, while the other measures humidity. Each room has four walls labeled according to the cardinal directions and a sensor can be affixed to exactly one of these walls. Additionally, the sensors are mounted at some height above the ground. In this example, although the units of measure for temperature, humidity, and height above the ground are not relevant, we will occasionally reference SI units.

The first term to consider is "metric". A metric is a thing that is being measured; for instance, in the example above, temperature and humidity are metrics because they are things being measured over time. A "measurement" is the value of a metric at a particular point in time. A measurement in our example could be 36.4°C for the metric "temperature". Unlike some other time series data systems, this model does not explicitly create support for the "multi-measure records" mentioned during our exploration of Amazon's Timestream service. For example, our model would not directly support a metric whose value is a two-dimensional vector, because this vector contains two values, or measurements. Additionally, our model does not envision the measurement as a string, but rather a numeric value. The simplest case we can use to illustrate this is the case of a GPS reporting its latitude and longitude over time. These two values are sometimes recorded as a unified string, like "31°52′38.3628″N 104°51′ 29.106″W". While our model does not provide outright support for this kind of time series, it does, however, offer the ability to record multiple measurements using our metadata feature, explained below and explored further in subsequent sections. Our metadata feature also allows for the recording of string data without further modification to this model, although using metadata for these purposes introduces some implementation complexities later on.

With this definition of a metric, the most basic time series example would be a series of measurements for a single metric over time. However, some data streams may contain multiple metrics but still be related to one another. For instance, in our example, the devices in the kitchen may produce different metrics (temperature vs. humidity), but they are clearly related regarding their general deployment location (in the kitchen). We believe it is natural in many scenarios to have multiple metrics which should be grouped together due to the relation of what they are measuring or even simply for organizational purposes. We call such a grouping of metrics an application, or "app" for short. Although the official definition we provide users with in our "Getting Started" guide states that an app is a collection of related data series, the definition of "related" is up for interpretation and can be defined by users to represent whatever is convenient for them and their situation.

Lastly, we need to define "metadata". Metadata refers to any additional information belonging to a measurement other than the metric name and the actual measurement value. In our example, metadata could include a label stating which wall the measurement was collected from, such as "south" for measurements collected by sensors located on the South wall, and a value, such as 54 cm, which describes how far off the ground the collecting sensor is mounted. Metadata in this model can be recorded on a per-measurement basis and there is no limit on the number of different metadata values which can be recorded, unlike some existing time series data solutions. Metadata is defined through a key/value structure, so in our running example, each data point would have a metric, like temperature, a value, like 36.4, and two metadata keys: location and height. Our model requires all measurements in a single app to have the same metadata keys; this is somewhat restrictive, but is the result of a compromise made to ease the process of ensuring that user queries for data can be answered quickly and that the data can be stored efficiently. We will explore some of the specifics of metadata in later sections.

```json
{
    "token": "abcdefghijk123",
    "app_name" : "office",
    "data" : "ewogICAgInRpbWUiOiAxNjQyMzgyMTEzLjkyNCwKICAgICJtZXR..."
}
```

Listing 1 A sample SIF packet.

4.1.2 Model data format

In addition to the definitions we provided above, we found it necessary to create a generic data format that is suitable for describing time series data sets in order to actually implement our model. This format contains three sections: an authentication token, an app name, and a data blob. Collectively, these three pieces of information constitute what we refer to as a "SIF packet", shown in Listing 1, because the packet contains all the details necessary to prepare data for ingestion into the SIF platform.

The purpose of the authentication token in our model format is twofold. First, the token identifies the data's owner at the moment when it arrives in the system. One of our system's goals is to improve overall usability and user experience, so it follows that users should not have to worry about how their data might interact with data other users are ingesting simultaneously. Thus, the authentication token allows us to separate data as it arrives according to its owner. Additionally, this token should also provide a layer of data integrity; a proper authentication token should not be forgeable, therefore it should remove the ability of malicious actors to insert arbitrary data on the behalf of legitimate users and poison their data sets. This token is included in our model based on the assumption that multiple users will want to use the system at the same time, an assumption which stems from the observation of our current time series data management system that accepts data input from a number of sources owned by varying parties. In this current system, steps have to be taken to ensure data does not mix across different users and/or sensors; here the token stands to improve things by making it possible for anyone to send in data without needing

to consider global uniqueness in any way.

In the model's data format, the app name is used to identify the application the incoming data should be associated with. When describing the vocabulary, we discussed how an application represents a user-defined grouping of metrics. Now, we see how the data format allows for an application to be explicitly defined for new data. Current systems also have some manual element of defining where data should be stored, such as a table name or similar database-specific location, so we do not believe continuing to require this information in the form of an app name as part of our new data format impacts user experience or the greater usability or scalability of the model.

The final portion of the model data format is a data blob. We call this segment a blob because we don't want to over-define what data should look like, since different sensors, devices, or other data sources may provide their information in different shapes, encodings, and according to other styles. Thus, in order to protect the user experience, our model does not constrain what user data should look like beyond the app name and token components. Accordingly, the data blob is restricted only to data which a program is able to decode into something representable using the set of vocabulary we previously outlined (i.e. it must be possible to transform the raw data into a time series expressable using the terminology established in Section 4.1.1). In the sample packet presented in Listing 1, we provide a partial Base64 string as data purely for concise demonstration purposes, not because Base64 strings are of special significance. Lastly, although we do not think it is necessary to define any specific data format that all users must adhere to, later we define a "standard" format which our implementation uses to ensure all components are speaking the same data language.

4.2 Decoupling from a database

The second fundamental design principle we considered is the ability for a time series data system to be decoupled from any specific database technology. In theory, it should not matter

how the data is stored; whether data goes to InfluxDB, a SQL server, an Excel spreadsheet, or something else, the system should operate in a way which allows for flexibility in the storage location. A potential criticism of this design principle is that one might wonder how often it would be necessary to change the underlying database once one has been selected. Although a database change may initially seem like a scenario which is rather unlikely to occur, we will demonstrate the value of this flexibility later on when discussing several key aspects of our implementation. Speaking more generally, the benefit of decoupling a time series data system from any one database technology allows the maintainers of that system to continually migrate towards databases which better support time series data. "Better support" could entail more effective data compression, speedier queries for time-sorted data, or any number of desirable traits. The decoupling we have outlined here allows the system designer to make their own choices about what underlying database is best for their scenario.

While existing systems are inseparable from their respective databases, our model aims to exploit the benefits of varying database technology to produce a modifiable and agile system model. For instance, with existing systems, any fundamental issue with the database technology produces an impasse; if, for some reason, the database is not suitable for the needs of a particular use case, the entire system must be avoided in order to skip over the problem. A real-world example of such a shortfall is with the open source time series database software OpenTSDB. In OpenTSDB, individual measurements are permitted to have variable metadata values. Since these metadata values are allowed to hold string data, it may initially appear that a user could use a system based on OpenTSDB to record data from a mobile temperature sensor and store the location of the sensor as metadata on the temperature measurement. However, OpenTSDB imposes a limit on the number of unique values an individual metadata field can take on, which means that after some period of time of storing this data, more measurements will be refused unless their recorded latitude and longitude have already been seen by the database. Although the limit on the number of unique metadata values is large, applications like GPS devices recording latitude and

longitude can reach it quickly given the improbability they frequently record the exact same position. Simply put, tying oneself to a single database makes it easy to inherit any flaws of that database, should they arise.

To briefly summarize the decoupling principle which guides our eventual implementation, we believe an effective time series data management system should not work only with a single database technology in order for the system to maintain long-term flexibility and upgradability. If a database service is rendered unsuitable for the purposes of a given use case, a system needs to be able to be shifted away to a new service without requiring a rewrite of a substantial proportion of the underlying infrastructure.

4.3 System-wide scalability

If a time series data management system existed which was perfect in nearly every way (blazing-fast queries, high data compression rates, supported any data format, etc), but did not scale well or at all, it would be completely unusable in many scenarios. Put simply, our third design principle is that the system must be scalable not merely in one or two aspects, but in every possible aspect. Examples of scalability range from the obvious to the subtle. For instance, one would expect the database component of a system to work regardless of use case, including working with both sparse and dense data sets without issue. These more obvious examples of scalability tend to revolve around user expectations of the system as it grows and becomes busier, however additional, more subtle, elements of scalability exist with regard to the effort a user must invest in order to get their data into the proposed system. For instance, consider the scenario where a user has a single device he/she would like to connect to the system. If the system requires two minutes of configuration on the user-end of things in order to get data flowing in from that device, that user will probably be satisfied. However, if some other user has a thousand devices that all need to be connected, he/she will likely be frustrated by how the device setup operation did not scale well and

requires him/her to invest over thirty-three hours of work to get the system configured for his/her devices.

We value scalability and demand its presence in all parts of our final implementation in order to avoid the scenarios described above. Ultimately the greatness of any system we build for time series data is irrelevant if it cannot handle a growing workload, especially considering the substantial growth in IoT many expect we will see in the coming years and how that will cascade down and result in new types and greater quantities of data-collecting sensors.

4.4 Zero-configuration data ingesting

The final major design principle which influenced our design process is somewhat related to the scalability issue we discussed above: a user might experience poor scalability through endless device configuration. While the configuration might be reasonable for between one and ten devices, it could easily become tedious and bothersome for tens or hundreds. In order to completely eliminate this phenomenon from the equation, we claim a desirable property of a time series data management system should be able to support data ingestion from a new device or data source with zero preconfiguration. For this purpose, we define preconfiguration to be any device or data source registration, definition, or provisioning, including, but not limited to, the manual creation of data buckets, tables, or other similar storage primitives, requiring any additional user input beyond the complete SIF packet itself in order for data to be successfully ingested, or any other pre-communication between the system and the user, excluding the negotiation of an authentication token, if required. In short, the user should only be required to send in data with a proper token in order for the system to attempt to process it. If the provided data is acceptable, it should be ingested and stored without requiring anything more from the user.

Having explained what we believe the premise of zero-configuration should include, we

must also describe what it should not include. First, zero-configuration does not mean users will have an issue-free experience when trying to configure their devices to stream data into the system. If a user does not adhere to certain guidelines and expectations laid out in the details of a specific implementation, such as an authentication token format or the expected structure of the SIF packet, the system cannot be reasonably expected to figure out the message's owner and their original intent. In anticipation of users making data input mistakes, the system is also not expected to always fix certain issues, like a user supplying the string "100" instead of the number one hundred. The zero-configuration design principle does not require this scenario to result in the system automatically converting "100" to one hundred. For the sake of usability and maintaining a simple user experience, the system may need to handle certain common issues when possible or make assumptions when an ambiguity appears, however this does not preclude the right of a system to fail to process a piece of data due to user error. In this vein, zero-configuration does not mean the system must always recover from all errors and proceed with data ingestion. Data is allowed to be determined invalid or un-ingestable. In order to make the user experience less frustrating compared to existing systems, the system should collect all errors which are most likely occurring due to user error and report them to the offending user. Some data ingest systems can irritate users when their data fails to reach the database but are not given a reason why. Consequently, zero-configuration will also imply that any new time series data management system includes some form of a warning system which alerts users to mistakes in their packets.

In summary, zero-configuration means a user should be able to stream data into the system without warning (i.e. no preconfiguration, registration, or definition) and errors, should they occur, need to be reported in a user-friendly manner which enables them to understand how and why the data failed to be ingested.

Chapter Five

SIF Design

Using the design principles outlined in the previous chapter, we created the Smart Infrastructure Foundation, or SIF, as our implementation of a modern time series data management service. SIF takes to heart the main ideas discussed above and combines them into a single cohesive technology stack that can be used to demonstrate the benefits of creating a system according to our new set of requirements. In this chapter, we will go over the product of our efforts, describing each component of the SIF platform, including its intended purpose, how it operates, how it satisfies the fundamental principles discussed in the previous chapter, as well as any implementation-specific details or requirements that came up.

5.1 Architectural overview

We first provide a brief overview of the system as a whole. SIF is implemented through six main pieces: the underlying database technology, the ingest component, the insertion component, a user-accessible API, a general management website aimed to wrap the API with a friendly visual interface, and data visualization tools. Properly ingested data lands in the database after the ingest component receives data from a user, authenticates and formats it, forwards it to the insertion component, and the insertion component generates and executes database insertion instructions. After data is in the database, it can be managed, retrieved,

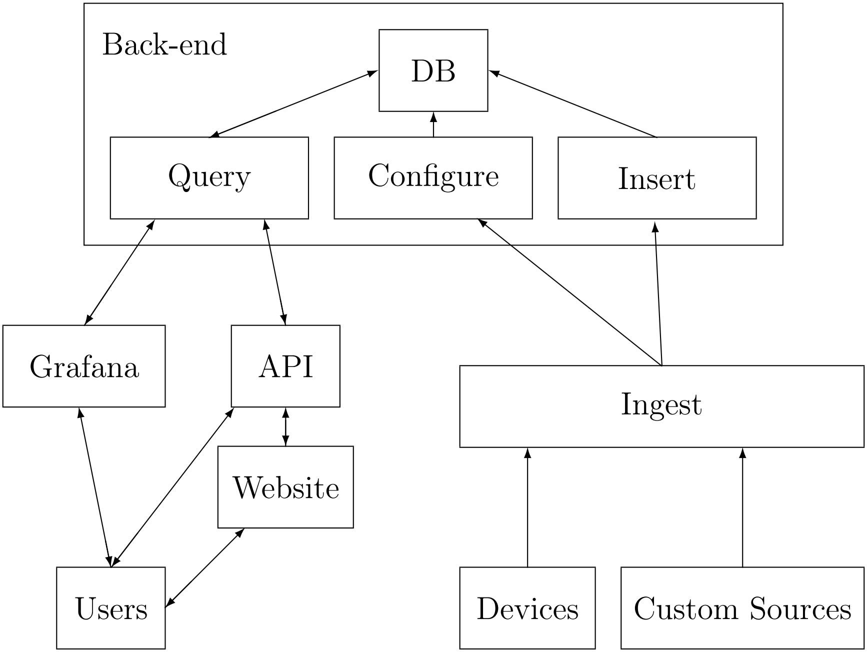


Figure 5.1 An architectural overview of the SIF platform

and visualized using the API, website, or our provided visualization tools.

5.2 Underlying database technology

The selection of a particular underlying database was the cornerstone of the implementation. We had to make sure our choice granted us enormous flexibility with respect to what we could store without sacrificing significant performance or losing substantial amounts of storage space. Taking into account the issues we saw with OpenTSDB and Optix.earth, we decided to use PostgreSQL as the bottom-layer database service [9]. PostgreSQL offers us the ability to store a practically unlimited number of unique metadata values for our measurements and many millions of rows of such measurements while maintaining fast query times. It also makes it easy to separate user apps by provisioning different tables. Different tables solve the issue of inter-user data collision by using a table naming scheme which makes it impossible for two users to insert data into the same table. In order to address the storage space concern, we opted to make use of a third-party PostgreSQL extension called

timestamp	metric	value	location	height
14 : 23 : 55	temperature	36.4	south	54
14 : 27 : 03	humidity	52.6	south	54
14 : 33 : 48	temperature	37.1	south	39
14 : 34 : 19	humidity	48.4	south	39

Table 5.1 An example of what raw data collected from the kitchen example might look like before compression occurs.

TimescaleDB [16]. Timescale is an optional add-on for PostgreSQL which offers a number of time series data-specific features. Notable features we make use of in this system include native data compression, optimization of time-sorted data queries, and improved row filtering performance when querying for rows with specific metadata values.

At a high level, Timescale's compression feature works by taking rows of the same metric which share metadata values and storing the timestamps and values as compressed lists within a single row. If we recall the example from earlier where the researcher had sensors in a kitchen and an office, their raw data might look something like Table 5.1.

Timescale compresses by operating on a number of *segmenting columns*, determined automatically in our system by the ingest component. Unique combinations of these columns define different rows in the compressed version of the table. The compression process squeezes data into a single row, storing the values of segmented columns (e.g. metric label and non-numeric metadata) only a single time. One implementation-specific detail is regarding numeric metadata, such as the height in this particular example. Since numeric data is stored as double precision values and only rows with equal values (e.g. rows with the location "south") are compacted together, using numeric columns as compression segments can result in smaller compression efficiency when floating point rounding errors occur. As a result, we follow Timescale's recommendation to avoid using floating point values as compression segments. Table 5.2 illustrates the behavior of segmented columns as opposed to regular ones. In the table, metric and location are segmented columns, while the others are regular timestamp and numeric columns.

By carefully creating an automated process for identifying segmenting columns, we also

timestamp	metric	value	location	height
[14 : 23 : 55, 14 : 33 : 48]	temperature	[36.4, 37.1]	south	[54, 54]
[14 : 27 : 03, 14 : 34 : 19]	humidity	[52.6, 48.4]	south	[39, 39]

Table 5.2 A compressed version of Table 5.1. The ingest component automatically configures TimescaleDB's compression algorithm on a per-app basis.

improve our query speeds. A common request a user might have for the system is to retrieve all temperature data recorded from sensors on the south wall of the kitchen. In this instance, PostgreSQL need only reference the table corresponding to the kitchen app; from this compressed table, a single row, shown above, contains all temperature readings under the desired circumstances. In order to answer the query, Timescale simply has to expand the compressed lists in this row instead of searching through a much larger set of rows looking for the desired metric and location values. While not all queries can be optimized this well, most see some benefit from this architecture.

5.3 Ingest component

The ingest component is the only part of the data ingestion pipeline which can be accessed by users. As mentioned previously, the ingest component, among other roles, serves three primary purposes: authenticate data arriving in the system, standardize its format, and pass it downstream to the insertion component. In our implementation, illustrated in Figure 5.2, the ingest component receives messages via the MQTT protocol, a lightweight message transmission service which operates on a publish/subscribe model [15]. In MQTT, clients wishing to send data first connect to a broker, then publish data to a particular channel, known as a topic. The ingest component hosts such a broker and creates a client of its own which subscribes, or listens, to all available topics. For each message that arrives, the ingest component spawns an asynchronous message handler to consume the received data.

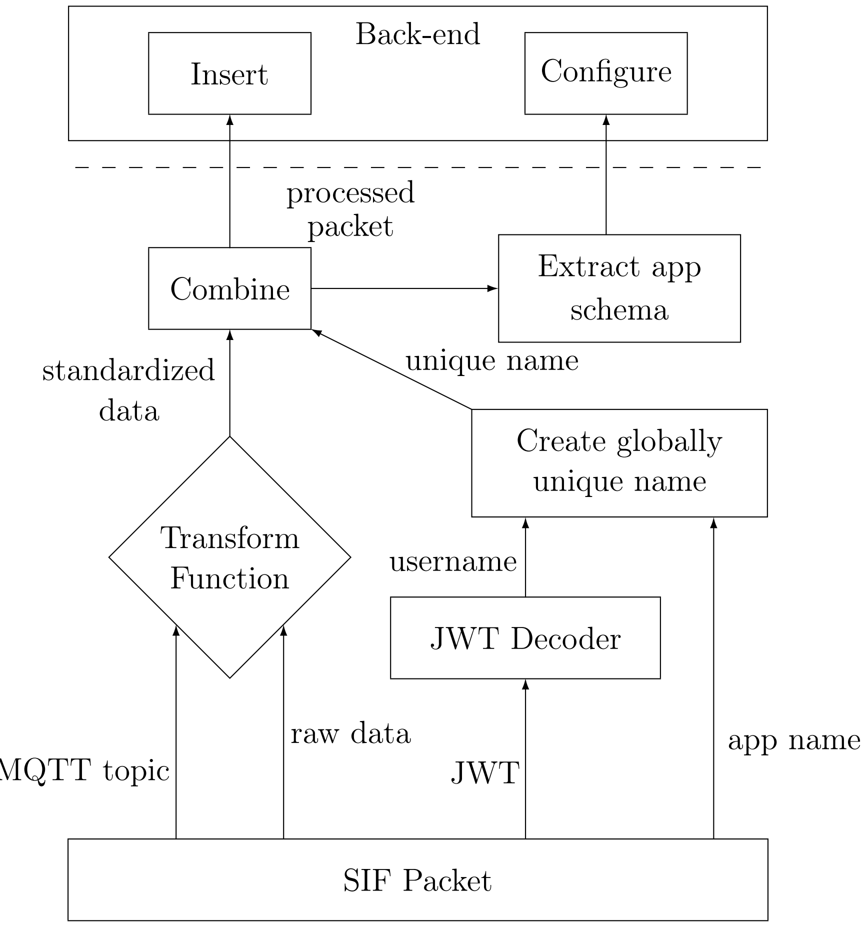


Figure 5.2 Internal design of the ingest component. Items above the dotted line are external to the ingest component and are included for clarity.

```
{
    "token": "abcdefghijk123",
    "app_name" : "office",
    "data" : {
        "time": 1642382113.924,
        "metadata": {
            "location": "south"
        },
        "payload": {
            "temperature": 28.3
        }
    }
}
```

Listing 2 Another sample SIF packet, this time using the standard SIF data blob format.

During this consumption process, the ingest component's message handler takes several steps. First, the handler decodes the incoming message. We decided to make all incoming messages adhere to a standard format: a stringified JSON which contains the pieces of the SIF packet, as we described earlier, in key/value pairs. We opted to outsource the creation and management of authentication tokens to Amazon's Cognito service, which generates JSON web tokens, or JWTs. Our decision to use Cognito is done in order to make implementation easier later on when adding authenticated API and website layers to the technology stack. Using a Cognito library, the ingest component decodes and validates the incoming token. If the token is valid, the associated username is extracted and classified as the owner of all data contained within that same packet. Recall that in Section 4.1.2, we declared our model to be data-format agnostic. While this is a great feature for our users, it adds a layer of complexity to our system because we have to handle any arbitrary data format. We achieve this through the use of data transformation functions defined on a per-topic basis. Using JavaScript, a user can write a custom transformation function which translates whatever their data is into a standardized format, explained below, and attach that script to a particular ingest component topic. As an example, assume their custom transformation script is run on all packets received on topic "ABC". Then, when publishing data to the ingest component, the data represented in the format expected by their transformation function can be published to the "ABC" topic and the ingest component will automatically know how to transform the data it is given. In our implementation, we use a hash map to map topics to transformation functions.

Transformation functions are expected to do one of two things: output a chunk of data formatted according to our implementation-specific data format or throw a transformation error. Our standardized data format is a JSON object with three required keys and one optional one. Required keys include payload, metadata, and time. Payload is an object which contains arbitrarily many key/value pairings, where keys are metrics and values are measurements for that particular metric. In Listing 2, we can see the payload field has

a "temperature" key which maps to 28.3. This will eventually represent a temperature measurement of 28.3. Metadata is also an object and contains key/value pairs according to the various pieces of metadata which are valid for a given app. Whether or not metadata is valid for a given app is determined through a process described below. Sticking with the same example, we see the location of this measurement is the southern wall, denoted by "south". Finally, the time field represents the time at which the measurement was taken, formatted according to Unix time (the number of seconds since January 1st, 1970 at 12AM UTC). We elected to use Unix time for providing timestamps in order to avoid all the issues associated with representing dates and times using strings (e.g. strict formatting rules, time zones, and daylight savings). The final component of our standard data format is an optional "device" field. When specified, the system will report errors it encounters and include the offending device in its report. For users ingesting lots of data from many sources into a single app, this can help narrow down where the issue is coming from.

Immediately after the data transformation step, the handler checks with our database to see if the necessary table and schema have been created. For new user apps, there will be no app table in the database yet, so in order for ingests to succeed, it must be created. In order to abide by our zero-configuration principle, the ingest component must do this creation automatically and dynamically. Using the username parsed out of the decoded authentication token and the app name provided by the user, the ingest component is able to generate a globally unique table name. Notably, this naming scheme makes it impossible for unwanted data mixing to occur between multiple users' data. In order to avoid unintended mixing within data streams owned by the same user, that user must take steps to avoid reusing an app name. In order to define the table schema for PostgreSQL and select segmenting columns for the TimescaleDB extension, the ingest component also needs to figure out what metadata is being supplied for the given app. To do this, the ingest component looks at the metadata supplied in the first packet for a new app. Each metadata field is added to the app table as an optional (null-able) column, and metadata keys mapping to non-numeric

values are marked as segmenting columns. Recall that our implementation is able to support string data, but that all measurements are expected to be numeric. To make this work, the ingest component also looks in the payload field for any metrics which map to non-numeric values and adds those to the app table as optional columns, but marks them as normal, non-segmenting columns, optimizing the table for TimescaleDB's data compression features without requiring any configuration on the part of the user. Although this automatic process does not always result in the ideal compression rate or table schema for a particular use case, its automated nature and overall accuracy are a reasonable approximation thereof. After creating the necessary database schema, the ingest component caches the app name for an hour to reduce how often it queries the database for the app's existence. It's important to not cache the app name forever, though, because users may opt to delete apps in the future and recreate them with different schemas. In order to support this feature, the caching time must be finite and not extremely long; this results in periodic checks by the ingest component which verify whether an app still exists. It's worth noting here that these database interactions on the part of the ingest component do not necessarily violate the decoupled database principle we discussed earlier. While the ingest component's programming would change should the underlying database technology be changed, the logic itself remains the same and requires only a change of database-specific language. Since the ingest component would need only a surface-level rewrite of its logic in order to be effective with a new back-end database service, we believe it satisfies the principle of a decoupled database.

Another interesting feature supported by the ingest component is the polling of user-defined custom data sources. We will use a device from The Things Network (TTN) as an example of such a custom data source. Data from TTN devices can be published to a MQTT broker hosted by TTN, allowing third party clients to connect, subscribe, and receive data points as they are sent to TTN. Every five minutes, the ingest component asynchronously checks a list of user-defined data sources to see if any new sources have been added which it needs to create listeners for. In the case of a TTN MQTT data source,

the ingest component will automatically instantiate a listener using the source information supplied by the user (host, port, username, password, etc) and connect it. For messages received through the custom source, the ingest component assumes all data therein belongs to the user who registered the source. It automatically parses out information such as app name, timestamp, and device name, if they are provided. If any of these are unavailable, generic defaults are supplied in their place (e.g. "source1" for app name, now for timestamp, empty string for device). Data supplied through custom sources is expected to already be in our implementation-specific data format since it is more complicated to specify an ingest transformation in the case of a custom source and that TTN (and other similar source providers) offer built-in transformation functions that can be used to properly format outgoing data. As custom source data streams into the ingest component, it is able to skip the authentication and transformation steps since the source is already associated with a user on our platform and properly formatted. All that needs to be done is to check the database to ensure the required schema exists and forward the parsed packet to the insertion component.

If the ingest component encounters an error of any kind during data transformation, automatic schema creation, or anywhere else in its ingest process, it will try to log that error to the system with as much detail as possible, including the user, app name, and device the data is associated with. Unfortunately, in order to associate errors with a user, they must occur after token verification. Therefore, the only class of error the ingest component is unable to report are token verification failures, since there is no way to attribute the error to a particular user if the identity of the packet's sender cannot be proven. Other errors which are not reported to users are those which were not caused by them. Examples include database connection failures, network problems, or bugs in the system. Prior to logging an error, the ingest component checks to make sure the error it encountered is one a user can actually cause; if not, it hides the error from the user and instead reports it to the debugging console for an administrator to examine later.

5.4 Insertion component

After the ingest component has fulfilled its role, we have made reference to how it forwards the processed packets to the insertion component. In our implementation, the insertion component hosts a second MQTT broker that is accessible only by itself and the ingest component. Processed data packets are published to the insertion component by the ingest component as they are prepared and upon their arrival the insertion component begins translating them into database-specific insertion instructions. In our case, this means the insertion component generates SQL insert commands based on the format of each formatted packet. As we mentioned in the previous section during our discussion of the ingest component, this does not violate our decoupled database principle because the insertion component's logic works with any database service. It just needs to be told how to convert our standardized messages into database inserts, and such a translation places no reliance on a particular database technology.

In our implementation of the SIF platform, the insertion component attaches a listener to its MQTT broker and instantiates handlers whenever a message is received. Recall that the ingest component provides the name of the app table which each message needs to have its data inserted into. Using this information, the insertion component needs only to convert the time, metadata, and payload fields into insertion commands. Each key provided in the payload object corresponds to a single row which is inserted into our database; for each of these keys, the insertion component knows the timestamp, metric, and value. To complete each row, the metadata object is inspected and additional column values are added based on its contents. Here, we see how metadata supplied in the metadata object is applied to every metric provided in the payload object and why if a user wanted to record temperature and humidity measurement where one was tagged with the location "south" and the other "north", these measurements could not be placed in the same payload object in the same packet. The decision to optimize towards shared metadata is based on the assumption that

most data residing in a shared data packet probably shares its metadata with its neighboring measurements. For instance, if a single sensor collected five different values, each value would be tagged with the same location.

With the metadata added to each row, it can simply be inserted as expected. There is, however, one special case which must be handled. We mentioned earlier that string data is supported, but adding this support resulted in some implementation oddities. The main point of interest regarding these oddities is located here in the insertion component. The task it performs is as simple as described above when only numeric data is provided. When string data is provided in the payload object, the insertion component recognizes this and modifies the insertion command to take into account the fact that non-numeric data cannot be provided as the value in app tables in our database. It sets the metric name in the same way as before (based on the provided key in the payload), but assigns a dummy value of zero. It then adds a metadata column to the insertion command, if it was not already provided, with the same name as the metric and sets the value of this column to the provided string value. Since the string data is implemented in our database as a special metadata value, the insertion component makes a special exception to the "shared metadata" rule to ensure that only a single row is actually given the string value that was sent in.

Similar to the ingest component, the insertion component listens closely for certain errors and reports them back to users when they are relevant. Most of the issues the insertion component encounters are related to database errors like unexpected data types or data which did not quite match the schema of the table it was intended for. These and similar errors are identified and reported with a user-friendly variant of the error. Instead of reporting an error like "error at pgctl.c 78: unknown column 'location' ", the insertion component identifies the meaning of this error in the context of the SIF cloud platform and would instead report "invalid metadata". According to the error chart shown in Table 5.3, a user can quickly discover the general source of the problem. This table contains not merely a sample of possible error codes the system can generate, but actually a comprehensive list. Although

Error	Meaning
invalid app	An app with the specified name was not found. This means the system was unable to automatically create this app for you. Make sure you're providing valid messages.
invalid metadata	Metadata keys were provided that did not match the expected metadata schema for the specified app. Double check the metadata you are sending and revisit the metadata discussion above to make sure you're doing the right thing.
unexpected string (expected number)	A string was provided as a metric value or as a metadata value for a metadata column whose data type is a number.
unexpected Boolean (expected number)	A Boolean was provided as a metric value.
unexpected object (expected primitive)	An object was provided as a metric value or a metadata value, but these values should only ever be numbers or strings.
unsafe app name	App names should only contain alphanumeric characters and underscores. This error occurs if other characters are included.
invalid blob	Data blobs you send to the ingest component should contain data and app_name keys. If either is missing, this error will be triggered.

Table 5.3 The definitive list of errors that can be reported by the ingest and insertion components and their respective meanings.

too few error codes would certainly be a net negative (since it would be harder to narrow down the problem), each error in the error list allows a user trying to debug issues to narrow down the location of the issue to a fairly specific part of the messages they are sending.

5.5 API

The ingest and insertion components cover the details of how users of the SIF platform can get data into the system. Although important, getting data into the cloud is really only half of what is necessary for a data management platform. In addition to storing their data, users need to be able to view and export it, as well as have other helpful options such as listing their apps or counting the number of data points in a particular time range. We meet this requirement by building and making available to our users a REST API. Below we will explain the functions of the API and how it adheres to the fundamental principles we outlined in Chapter 4.

Our API offers a small but effective number of endpoints to access user data, allowing

users to: list the apps registered to their account, get the schema for an app which he/she owns, download data from a particular owned app for a given time range that match a given set of metadata key/value pairs, download only a single metric (e.g. temperature) from a particular owned app for a given time range that match a given set of metadata key/value pairs, get a list of all metrics in a given app which he/she owns, and delete any owned apps. A full API reference can be found on special documentation endpoints, linked in the references below [11, 12]. As hinted at by the continued references to app ownership, all API requests are authenticated using the same token scheme that we described when discussing the ingest component in Section 5.3. Using this token, the API automatically determines the user initiating the request and is able to run their queries only against apps which they own. Aside from ownership and authentication, the main component which one may believe to be missing from the API is the ability to filter downloads beyond that of a metric name. Although it is true that downloads do not offer any fine-grain filtering functions beyond metric names and basic metadata matching, we make the assumption that users will be able to use our visualization service to make such specific queries, while downloads are intended to be more of a bulk data extraction feature. Another point of note regarding the API's download feature is the use of a streamed response for all downloads. Notable benefits of this approach, this reduces the amount of memory the server hosting the API request must have and provides users with the ability to see the data arrive as it is extracted. However, these positives come at the cost of an increased number of SQL queries made to our database and a degradation of query speed as a download progresses through a time range containing many data points (somewhere on the order of $10^{6.8}$ measurements in a single time range). Since most use cases will either be where a much smaller number of data points are being downloaded or where large downloads are not expected to finish quickly, and that downloads larger than 1GB could crash the server hosting the request, we believe the streamed response to be an overall positive.

Regardless of its features, the API must align with the design principles we outlined

My Apps

The download button in each row of the apps table, below, will start a download for that app's data across a particular time range. Set the bounds of that range here.

All times are relative to your local time zone: EST (UTC-5)

Range Start 01/31/2022 07:32:26

Range End 01/31/2022 13:32:26

Name	Schema	Metrics	Download	Delete
machineUsage	Metadata: machineName — Data Type: TEXT	netOut, cpuUsage, netIn	DOWNLOAD (1,173)	DELETE
vst0	CLICK TO REVEAL	CLICK TO REVEAL	DOWNLOAD (0)	DELETE
vst1	CLICK TO REVEAL	CLICK TO REVEAL	DOWNLOAD (0)	DELETE

Figure 5.3 After a user has logged into the SIF website, they are greeted with a landing page which provides a list of their current apps.

earlier. Although the API cannot necessarily decouple itself from the database due to its role as a data-extractor, it was written in such a way which requires only the changing of database connection parameters and queries to work with a new data source. On the topic of scalability, most of the API endpoints operate within Amazon's API Gateway, a server-less API architecture which permits massive scaling by instantiating additional request handlers on-demand. As of writing, it is not possible to use a streamed response with Lambda functions in the API Gateway, so the download endpoints are hosted on a separate EC2 instance placed under an application load balancer. The load balancer allows the download endpoints to scale up, if necessary, by replicating the EC2 container and attaching it to the load balancer as an alternative request handler.

5.6 Website

The API is a great feature available to users with knowledge of how to use such a service, but for users interested solely in basic data management and visualization, we also created a website which wraps the API with simple features like showing a list of apps, initiating data downloads from certain apps across a given time range, registering custom data sources, viewing the user's error log, and more. The website is hosted in the same scalable fashion as the download endpoints of the API, providing for easy scaling if many users need to visit the website simultaneously. A link to visit the website is provided in the references section below [13].

Upon visiting the website, users must first login through Amazon's Cognito service in order to generate a token to use for authenticating future requests to the API. Once logged in, the user is greeted with a list of their apps, shown in Figure 5.3. For each app, the schema and metric list can be dynamically loaded, data can be downloaded from the specified time range, and the app itself can be deleted. Using a navigation bar, users can access the custom data source registration panel and input the details necessary to get data from a TTN application into the SIF platform without programming anything. Finally, users can view a list of errors the ingest and insertion components have encountered which were attributable to them.

5.7 Data visualization tools

Although time series data can be easy to discuss, we find that it is often the case that a visualization of what is being discussed conveys the ideas at hand much faster and more reliably than pure text or speech. Accordingly, we elected to make Grafana, an industry standard data visualization tool, available to our users. Getting new users set up with Grafana requires minutes and, once their account is ready, they can immediately start viewing data

being ingested in real time without needing to configure any database connections. Those familiar with Grafana may be aware of the process required to add a new data source and how it can be tedious if all the database connection details are not readily available. In order to protect the security of our database, we keep these details private and simply make a shared data source available to all our Grafana users. This shared connection is limited to viewing data, so it's impossible to accidentally harm the integrity of any stored data while trying to make visualizations.

One drawback of our shared connection approach is the inability to regulate access to certain data in our database. Specifically, any Grafana user is able to view data contained in any app table, even those which they do not own. This is the primary limitation of taking the shared database approach, and can be mitigated using several mechanisms, one of which we describe below. In our case, this is not a major problem because sensitive data is not being stored and the purpose of our user authentication steps is primarily organizational and to prevent unintentional data-mixing between users. If a different use case required more data security in Grafana (i.e. users can only see their own apps), the user setup process would be more complex because a shared connection could not be used. Instead, the ingest component would need to generate and issue database access credentials (username and password) to each user when they create their first app. These credentials would only be permitted to view app tables owned by the user they were issued to. Then, in Grafana, that user would need to input all of our database connection information (e.g. host, port, etc.) followed by the database username and password generated by the ingest component. We determined that this additional complexity was, in our case, not worth the benefit of increased data security. In the case where such security is desired, the process we just described would be our first path forwards.

Chapter Six

Evaluation

After working on our implementation of the SIF platform, we evaluated it by having two users with very different time series data sets try to ingest their data. The tests occurred when our implementation was finished but open to revision based on user feedback. Through these experiments we learned what worked, which features were most helpful, what stood to be improved, and the things our test users did not enjoy so much. In this chapter we cover these topics and how the system has evolved as a result of input from our testing audience.

6.1 Dense and historical data

Our first test aimed to demonstrate the system was actually usable by someone without insider knowledge of what was going on, could easily and seamlessly handle prerecorded data, and that our data output services (our connection to Grafana and the user API) successfully managed to deal with very dense data sets.

With respect to usability, the SIF platform proved very capable. With the appropriate documentation, our first test user was able to easily modify a sample data ingestion script we provide in a "Getting Started" guide and get his data flowing towards the ingest component. We include a copy of the "Getting Started" guide, also known as the "SIF User's Handbook", in Appendix A. We observed the ingest component to ensure it created the database schema

we expected based on the data that was being sent in. Our automated mechanisms for detecting and building the appropriate table schema, selecting columns to use as compression segments, and transforming data into our standard intermediate format all worked as expected. Recall from Section 3.1 where we explored how Amazon's Timestream service had, at times, long delays associated with the ingestion of past data due to the in-memory versus magnetic storage decisions being made in the background. SIF demonstrated the ability to ingest past data points with no delays, even in the event data needs to be inserted into a region of a table that Timescale has already compressed.

6.2 Real-time, diverse, and metadata-rich data

In the second testing scenario, we wanted to demonstrate the system's ability to handle real-time collection of data that is currently being processed and stored by our lab's existing time series data management system. Unlike the first test, the data collected by our lab has lots of associated metadata and packets with highly variable schemas, providing a more rigorous set of tests on the ingest and insertion components' abilities to handle dynamic environments. The ingest component continued to validate the automated mechanisms that determine compression segments and table schemas. Meanwhile the insertion component proved its database query translations could handle many permutations of our standard data format. As in the first case, the process of getting the data flowing into our system was relatively smooth, although this phase illuminated some problems that needed to be addressed. At the time of testing, the SIF platform lacked the ability to report errors to users, which resulted in some token and data formatting problems. These issues inspired us to find a solution to the problem of error reporting and develop the scheme described in Section 5.3.

6.3 A custom data source

To test the custom data source feature, we used a LoRaWAN GPS device and routed its data through the Things Network (TTN). A custom transforming function on TTN was used to translate data posted by the device into the format expected by the SIF platform from custom sources. Using the website, we created a custom source to link our TTN app to a SIF app; after that, the ingest component automatically started polling TTN's MQTT broker for GPS data and routed it to the rest of our ingestion pipeline. One issue we encountered during this process was the delay associated with adding or deleting a custom source from an account. When a new source is added, it can take up to five minutes for the ingest component to realize, since it checks the list on a set interval. Similarly, a deleted source can take up to five minutes to actually stop ingesting data, despite it being removed from the database immediately.

6.4 A survey of the system in the context of our goals

The final point of evaluation is to have a brief discussion around the system's overall performance, how well the model performed in helping us guide the implementation, and if there are any improvements which could be made to our design model moving forwards. Across all tests, we ingested around six-hundred million data points. Before the compression agent ran, this data consumed over fifty gigabytes due to the unnecessary repetition of metric names and static metadata items. However, after the Timescale compression agent ran, this consumed less than three gigabytes on disk, implying less than five bytes of storage per data point and over a ninety-five percent reduction in storage consumption. One limitation we discovered in this regard was the periodic nature of the Timescale compression agent. The agent runs on a set interval (for SIF, this interval is seven days) and compresses data points which have been added since the previous execution. As a result, the disk where the

database is located must be large enough to contain one interval worth of uncompressed data. For some applications, like ours, this could be significantly larger than the amount of space required to store the compressed versions. Regardless of this limitation, though, this result proved our automated methods of detecting data schema and configuring the compression agent accordingly are generally effective at reducing the space required to store data without asking the user to set up anything. In a more general sense, other system components like the API, visualization tools, and the website all performed well and proved to be helpful tools in all of the testing scenarios we described above. Finally, the model was crucial to guiding the overall development of the SIF platform. In particular, we found our specification of a standard data format and the need for zero-configuration ingestion to be of special importance in identifying where our system should begin (start with standardization of data) and what it needs to do (figure out how to store what it is sent).

Chapter Seven

Future work

There are at least three immediate avenues for future improvement of the SIF platform. The first of these improvements, metadata malleability, makes the system more closely aligned with the zero-configuration design principle we outlined in Section 4.4.

7.1 Metadata malleability

When the ingest component generates a new app table, it uses the first data packet's schema to dictate that of the entire app table. This is sufficient in most use cases, however we recognize that the dynamic and automatic addition of new metadata columns at arbitrary points in time would be a helpful feature for some users. Such a feature raises certain questions; for example, what is the best way to backfill the new column for existing data points (e.g. use null, specify a default, or a more complex mechanism)? Further research and experimentation is necessary to determine whether Timescale can be instructed to handle such table alterations efficiently – or at all – particularly when such alterations occur on compressed tables. In order to adhere to our decoupled database principle, any issues with Timescale in this regard would need to be resolved without impacting users.

7.2 Security improvements

As it stands, the SIF platform is great for demonstrating many of the features we would like to see in more time series data management systems, however it is not necessarily ready for a production-level deployment. In particular, there are at least two security-related improvements which could be made in the future. First and foremost, we highlight our use of the MQTT protocol. In practice, we would use MQTT over SSL to ensure security and privacy of data while it is being transmitted to the ingest component. Since security was not a major focal point of the SIF project, we did not use SSL with our MQTT brokers in order to simplify the implementation. Second, access to our time series database could be more strictly controlled. As it stands, users of the SIF Grafana can technically view data from others' applications since all of our Grafana accounts use a shared PostgreSQL user. This is a consequence of using a shared connection. In practice, we suggest adding a feature that allows SIF to automatically create database users that only have access to apps belonging to a particular person and sharing those credentials with the corresponding SIF user via the website. We did not pursue this in our sample implementation since it would require each user to create a database connection, potentially involving the configuration of SSL certificates, a process we decided was not necessary in our case.

7.3 MQTT broker software

The SIF platform makes use of the open source MQTT broker software "mosquitto" [7]. Throughout our tests and use of the system, the mosquitto broker proved perfectly capable in all regards except one. In order to make scalability of the ingest and insertion components easier, shared MQTT topics are highly desirable. Shared topics allow multiple clients to subscribe to a single topic and have messages published to that topic spread evenly instead of each client being issued a copy. This automated load balancing mechanism is perfect

for our architecture since multiple copies of the ingest and insertion components' processes could be launched and they would all share the work evenly. Unfortunately, we discovered mosquitto has not been updated to support shared topics, as it is a relatively new addition to the MQTT protocol. In the meantime, we implemented a manual form of load balancing where the ingest component splits its output messages across different "channels" on a topic. For example, the topic "/data" with channels 0 and 1 becomes topics "/data/0" and "/data/1". Once support for shared topic subscriptions becomes more standard, we recommend moving away from the manual balancing used in our implementation.

Chapter Eight

Conclusion

At the outset of this paper, we described some of the issues which existing time series data management platforms often face. In particular, they are usually designed with a preference towards the underlying database or towards its target audience, and this design focus tends to result in obvious shortfalls on the side not primarily considered. We aim to help alleviate these problems by proposing a new design framework to guide the construction of time series data management systems. Then, using this framework, we build an end-to-end time series data management platform which aims to solve some of the issues users face when trying to use such data storage services in practice. We evaluate the implementation with three distinct time series data use cases, analyzing the parts of the system which succeeded and those which stand to be improved. Finally, we offer some potential future directions based on what we have learned throughout this process.

Based on our work and results, we believe time series databases as they currently exist are capable of supporting new use cases, but often require the development of substantial surrounding software infrastructure. These modifications often introduce new design trade-offs, like balancing back-end performance and front-end usability, which do not have straightforward answers and can be time consuming issues to deal with. We believe our proposed data model and design principles aid in the design of systems which seek to better balance these concerns and that our sample implementation of a platform built using such principles

demonstrates how an effective blend of performance and usability can benefit the system, its administrators, and its users.

REFERENCES

[1] Amazon. *Timestream*. Amazon Web Services, 2020. URL: https://aws.amazon.com/timestream/.

[2] Ala Arman et al. “Automating IoT Data Ingestion Enabling Visual Representation”. In: *Sensors* 21.24 (2021), p. 8429. DOI: 10.3390/s21248429.

[3] Michael Corsello, Zach Johnson, and Joel Morgan. *Optix.earth*. Charlottesville, Virginia: General Atomics Commonwealth Computer Research, Inc, 2020. URL: https://www.optix.earth/.

[4] Luca Deri, Simone Mainardi, and Francesco Fusco. “tsdb: A compressed database for time series”. In: *International Workshop on Traffic Monitoring and Analysis*. Springer. 2012, pp. 143–156.

[5] Shushan Hu et al. “Building performance optimisation: A hybrid architecture for the integration of contextual information and time-series data”. In: *Automation in Construction* 70 (2016), pp. 51–61. DOI: 10.1016/j.autcon.2016.05.018.

[6] John Meehan et al. “Data Ingestion for the Connected World.” In: *CIDR*. 2017.

[7] *Mosquitto*. Eclipse, 2009. URL: https://mosquitto.org/.

[8] Tuomas Pelkonen et al. “Gorilla: A fast, scalable, in-memory time series database”. In: *Proceedings of the VLDB Endowment* 8.12 (2015), pp. 1816–1827.

[9] *PostgreSQL*. The PostgreSQL Global Development Group, 1996. URL: https://www.postgresql.org/.

[10] Sean Rhea et al. “Littletable: A time-series database and its uses”. In: *Proceedings of the 2017 ACM International Conference on Management of Data*. 2017, pp. 125–138. DOI: 10.1145/3035918.3056102.

[11] *SIF API*. 2022. URL: https://api.uvasif.org/v2/documentation.

[12] *SIF Download API*. 2022. URL: https://download.uvasif.org/v1/documentation.

[13] *SIF Website*. 2022. URL: https://uvasif.org/.

[14] Satyajit Sinha. “State of IoT 2021: Number of connected IoT devices growing 9% to 12.3 billion globally, cellular IoT now surpassing 2 billion”. In: *IOT Analytics* 1.4 (Sept. 2021).

[15] Anthony Stanford-Clark and Arlen Nipper. *MQTT*. MQTT.org, 1999. URL: https://mqtt.org/.

[16] *TimescaleDB*. Delaware, United States: TimescaleDB, Inc., 2018. URL: https://www.timescale.com/.

[17] *Architecture - Amazon Timestream*. 2020. URL: https://docs.aws.amazon.com/timestream/latest/developerguide/architecture.html.

[18] *Timestream Concepts*. 2022. URL: https://docs.aws.amazon.com/timestream/latest/developerguide/concepts.html.

[19] *Writes*. 2022. URL: https://docs.aws.amazon.com/timestream/latest/developerguide/writes.html.

[20] *Smart Infrastructure | Strategic Investment Fund*. 2022. URL: https://sif.virginia.edu/smart-infrastructure.

[21] Chen Wang et al. “Apache IoTDB: time-series database for internet of things”. In: *Proceedings of the VLDB Endowment* 13.12 (2020), pp. 2901–2904. DOI: 10.14778/3415478.3415504.

[22] Yang Yang, Qiang Cao, and Hong Jiang. “EdgeDB: An efficient time-series database for edge computing”. In: *IEEE Access* 7 (2019), pp. 142295–142307. DOI: 10.1109/ACCESS.2019.2943876.

APPENDIX

Appendix A

SIF User's Handbook

Beginning on the next page, we include the SIF User's Handbook, also known as the "Getting Started" guide. This document was provided to our system testers in Chapter 6.

SIF User's Handbook

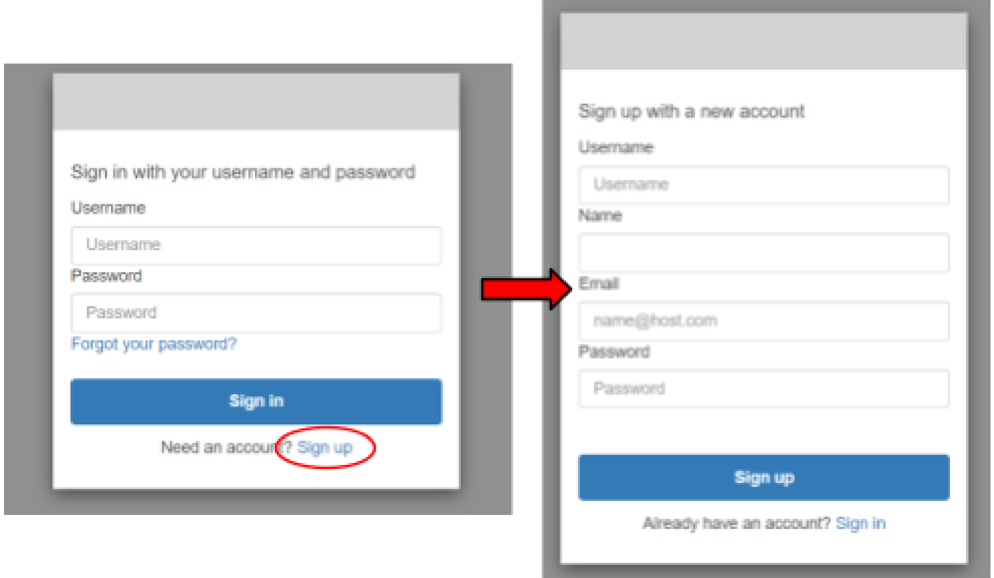
Written by G. Michael Fitzgerald II

Last edited April 27th, 2022

The comprehensive source for getting started with the SIF cloud platform

I. Creating a SIF account

Visit https://uvasif.org and select “Sign Up”. Create a username and password, and enter your name and email into the form. You need not enter your full name; a first name, nickname, or initials are all fine. You will need this username and password to access all portions of the SIF platform later on.



Verify your account was created successfully by signing in. If successful, you will be greeted with the screen shown below. You will use the same username and password here in order for the website to generate an access token for the current session.

We understand logging in twice seems redundant. Here is a brief explanation of why: we use an Amazon authenticator to control access to the website. When you sign in, the authenticator provides the website with an *access* token, not an *identity* token. Our API requires an *identity* token, so we are left to generate this on our own. In practice, this means the first login grants you access to the website, and the second login dictates whose apps you see. This allows you, the user, to switch between different “profiles” without having to globally log out each time. It also allows us to make sure you don’t get signed out unexpectedly by refreshing the token before it has a chance to expire.

Enter your API access credentials.

Re-entering your credentials here allows you to change API access permissions without logging out of the website.

Username

Password

AUTHORIZE

After authorizing, you should be greeted by a screen informing you that you have no apps. If you see this, you're done with this step and should move on to step two. If anything else happens, contact a system administrator.

II. Creating your first app

Apps are created automatically as data is sent into the system, so let's get set up with sending data. This example uses Javascript, but any language with MQTT and Cognito libraries would work. Other SIF users have also had success with Python.

→ **Step 1:** Generate a Cognito *identity* token

In Javascript, we can use the amazon-cognito-identity-js package for this task. Here's a snippet of example code:

```
1  const ACI = require("amazon-cognito-identity-js");
2  var idToken;
3
4  const authDetails = new ACI.AuthenticationDetails({
5      Username: YOUR_USERNAME,
6      Password: YOUR_PASSWORD
7  });
8  const cognitoUser = new ACI.CognitoUser({
9      Username: YOUR_USERNAME,
10     Pool: new ACI.CognitoUserPool({
11         UserPoolId: "us-east-1_yfAGwxbYW",
12         ClientId: "4bfuvavalple0k8k6lj4oln5ne"
13     })
14 });
15
16 async function refreshToken() {
17     cognitoUser.authenticateUser(
18         authDetails,
19         {
20             onSuccess: function(result) {
21                 const token= result.getIdToken().getJwtToken();
22                 console.log(
23                     "[%d] refreshed token: %s", new
24                     Date().getTime() / 1000,
25                     JSON.stringify(token)
26                 );
27                 idToken = token;
28             },
29             onFailure: function(err) {
30                 console.error(err);
31             }
32         }
33     );
34 }
```

→ **Step 2:** Generate (or collect) some data

Any data format is supportable by the SIF platform. It is just a matter of defining your format and getting a system administrator to add a new transformation topic for you. Alternatively, you can bypass this step immediately if you're willing to adhere to a particular format. This default SIF format is a JSON, structured as follows:

```
{
    "time": 1643298089.292, // Unix epoch time (seconds, not millis)
    "device": "", // optional string to identify devices in the error log
    "metadata": { // optional per-datapoint metadata key/value pairs
        "location": "south_wall",
        "facing": "east"
    },
    "payload": { // key-value pairs for each measurement
        "metric1": 0.00,
        "metric2": 0.00
    }
}
```

With this format, one datapoint is created for each metric in the payload. When multiple metrics are present, they are all given the same timestamp and metadata. If individual data points require different timestamps and/or metadata, they will need to be contained within separate instances of this JSON. If your data is in the default SIF format, here is an example of how you can send it to our system to be stored:

```
1  const mqtt = require("mqtt");
2  const client = mqtt.connect("mqtt://broker.uvasif.org");
3
4  async function collectData() {
5      try {
6          const timeval = new Date().getTime() / 1000;
7          const packet = {
8              time: timeval,
9              metadata: {
10                 sensorType: "example",
11                 fiftyFifty: parseInt(timeval) % 2
12             },
13             payload: {
14                 sampleMetricA: Math.sin(timeval / 4),
15                 sampleMetricB: Math.cos(timeval / 4)
16             }
17         };
18         const blob = {
19             app_name: "exampleApp",
20             token: idToken,
21             data: packet
22         };
23         console.log("[%d] Publishing data", timeval);
24         client.publish("data/ingest/passthrough", JSON.stringify(blob));
25     } catch(err) {
26         console.error(err);
27     }
28 }
```

Note the variable "blob" defined on lines 18-22 in the example above. This blob format is **required for all data** sent to the SIF platform. Blobs received that have invalid tokens, do not have an app name, are missing a data key, or any combination of similar issues are **rejected immediately**. Your top level data structure must be like this blob: a stringified JSON with token, app_name, and data keys.

There is one important thing to mention with respect to metadata. When a blob is sent in with a new app name, the system recognizes this and automatically creates the new app **according to the metadata schema present <u>in that blob</u>**. This means that if the first blob with app name "myFirstApp" has no metadata keys, any future blob with app name "myFirstApp" with metadata keys will be rejected. In short, the metadata keys present in an app's first blob dictate the full set of valid metadata keys. Future blobs for that app need not present all of the metadata keys in order to be accepted, as each metadata key is optional, but no blob may present any metadata key that was not present in the first blob. Furthermore, the data type of a metadata value is also dictated by the first blob. Metadata values can be either strings or floats, and cannot switch between the two.

<u>Examples:</u> Assume all blobs in this example are sent in with the app name "abc". The first blob has the following metadata:

```
{
    "location": "kitchen",
    "manufacturer": "ACME",
    "deviceId": 0
}
```

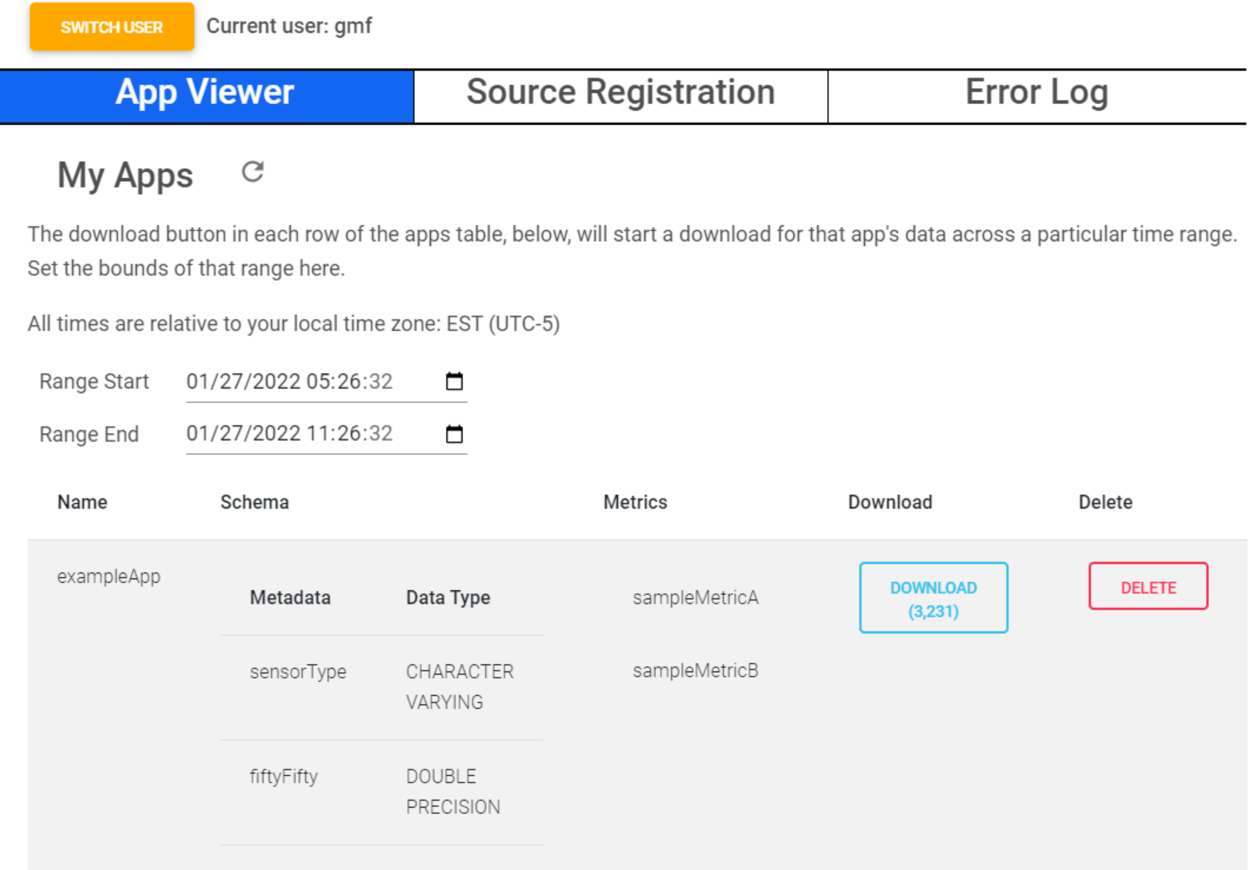
For each of the following subsequent blobs, we provide the metadata section and assert whether this would be accepted or rejected by the SIF platform:

Metadata	Accepted or Rejected?
`{ // no metadata keys }`	ACCEPTED
`{ // valid subset of the initially provided keys "location": "kitchen" }`	ACCEPTED
`{ // valid subset w/ incorrect data type "location": "office", "deviceId": "zero" }`	REJECTED
`{ // invalid subset "orientation": "left", "city": "New York", "height": 73 }`	REJECTED

III. Managing your apps at https://uvasif.org

A. Using the App Viewer

After you have sent in some data, log back into the website and a new app should have appeared.

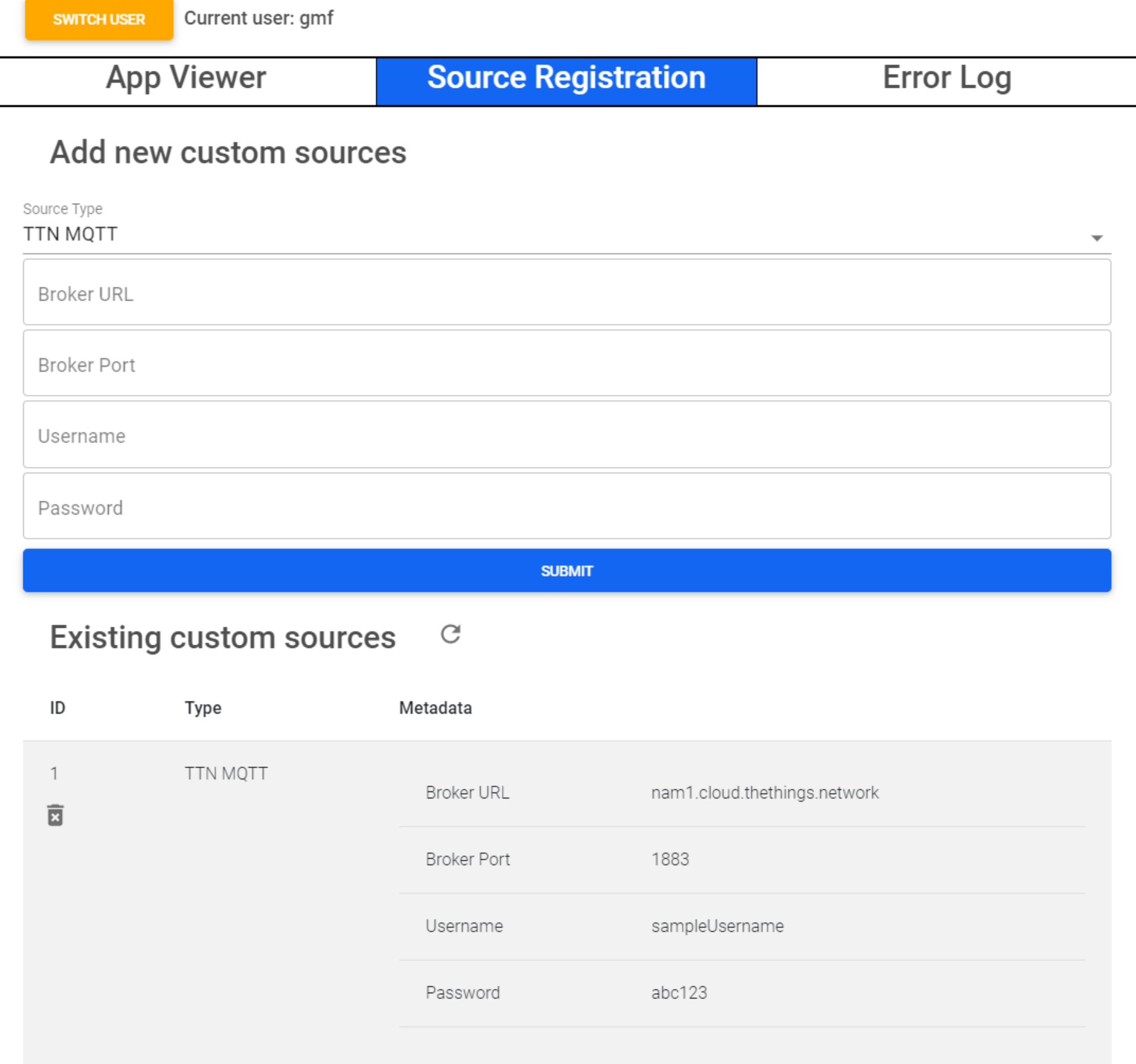


Here we can see the exampleApp has appeared and we are presented with some information about the app. In the schema column, we are told the metadata schema of this app. In the metrics column, we see a list of all metrics that have ever been received for that app. We are also presented with two buttons: download and delete. The download button will download all data for the app in the time range specified above the table. The number of data points in this range is shown in parentheses on the download button (in the image, there are 3,231 data points in the given time range). The delete button will **delete the entire app and all of its data forever**. After deleting an app, you may need to refresh the page in order for the app to actually disappear.

B. Registering custom sources

If you have an existing data stream that you'd like to redirect towards SIF, then custom source registration is built for you. In the SIF website, you can pick a type of custom data source

(as of writing, support is limited to TTN-MQTT (The Things Network) streams), supply any necessary credentials, and you're done. Every 5 minutes SIF updates the custom source list and attaches listeners to those added by users.



A major point of note with the custom data source system is expected data format. The ingest system cannot currently interpret different formats originating from custom sources, so any and all data published to a custom source and ingested into SIF must be structured according to the standard SIF data format described previously in step 2 of Section II. For TTN-MQTT streams, a custom formatter should be deployed with your application which outputs a SIF-standard data blob in the decoded_payload section.

Data collected from custom sources will be filed under app names according to the following logic: if the data packet contains an app_name field, use it as the app name. If such a field is not present, use "customX", where X is the custom source ID, as the app name. For

TTN-MQTT data, the system will try to extract the name of the TTN application where the data is from and use that if an app_name field is not present.

C. Viewing the error log

Select “Error Log” from the navigation bar at the top of the website. If the system encounters problems when processing data, it will try to associate that data with its owner and log the event. The ingest system is capable of detecting several different common error types, shown in the table below. In order for an error to appear in this log, it must have occurred in a location where the system had access to your user name; this means that if you provide an invalid token, that error will not appear in the log since the system was unable to verify the owner of the token.

SWITCH USER Current user: gmf

App Viewer | Source Registration | Error Log

Recent Errors

If a device name is available for a given error, it will be shown in (parentheses) next to the app.

ID	Timestamp (Local)	App (device)	Error
10	3/28/2022, 10:12:50 PM 5 minutes ago	sampleApp (deviceXYZ)	assertion failure: received empty payload (no metrics)
23	3/28/2022, 9:51:47 PM 26 minutes ago	sampleApp (deviceABC)	assertion failure: received empty payload (no metrics)
266	3/9/2022, 3:19:38 PM 3 weeks ago	machineUsage	invalid metadata

By default, the “device” column in the error log is blank. If you are experiencing many errors or need to trace an error back to a single device, use the “device” field in the standard SIF data format. When a loggable error event occurs, the system will log the device alongside the error message, if one is provided.

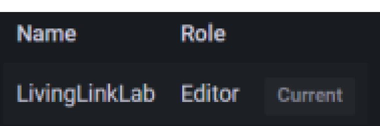
Entries in the error log are unique with respect to the (app_name, device, error) tuple. The timestamp of each entry represents the most recent time at which that error was triggered on the specified app and device.

Errors & Their Meanings

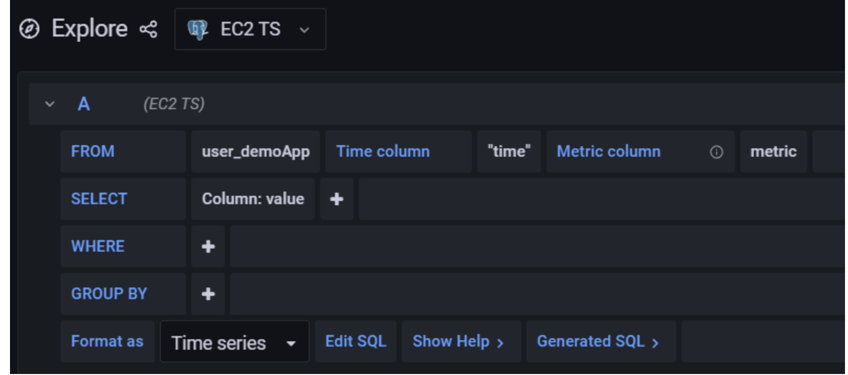
Error	Meaning
invalid app	An app with the specified name was not found. This means the system was unable to automatically create this app for you. Make sure you're providing valid messages.
invalid metadata	Metadata keys were provided that did not match the expected metadata schema for the specified app. Double check the metadata you are sending and revisit the metadata discussion above to make sure you're doing the right thing.
unexpected string (expected number)	A string was provided as a metric value or as a metadata value for a metadata column whose data type is a number.
unexpected boolean (expected number)	A boolean was provided as a metric value.
unexpected object (expected primitive)	An object was provided as a metric value or a metadata value, but these values should only ever be numbers or strings.
unsafe app name	App names should only contain alphanumeric characters and underscores. This error occurs if other characters are included.
invalid blob	Data blobs you send to the ingest broker should contain data and app_name keys. If either is missing, this error will be triggered.

IV. Visualizing your data on Grafana

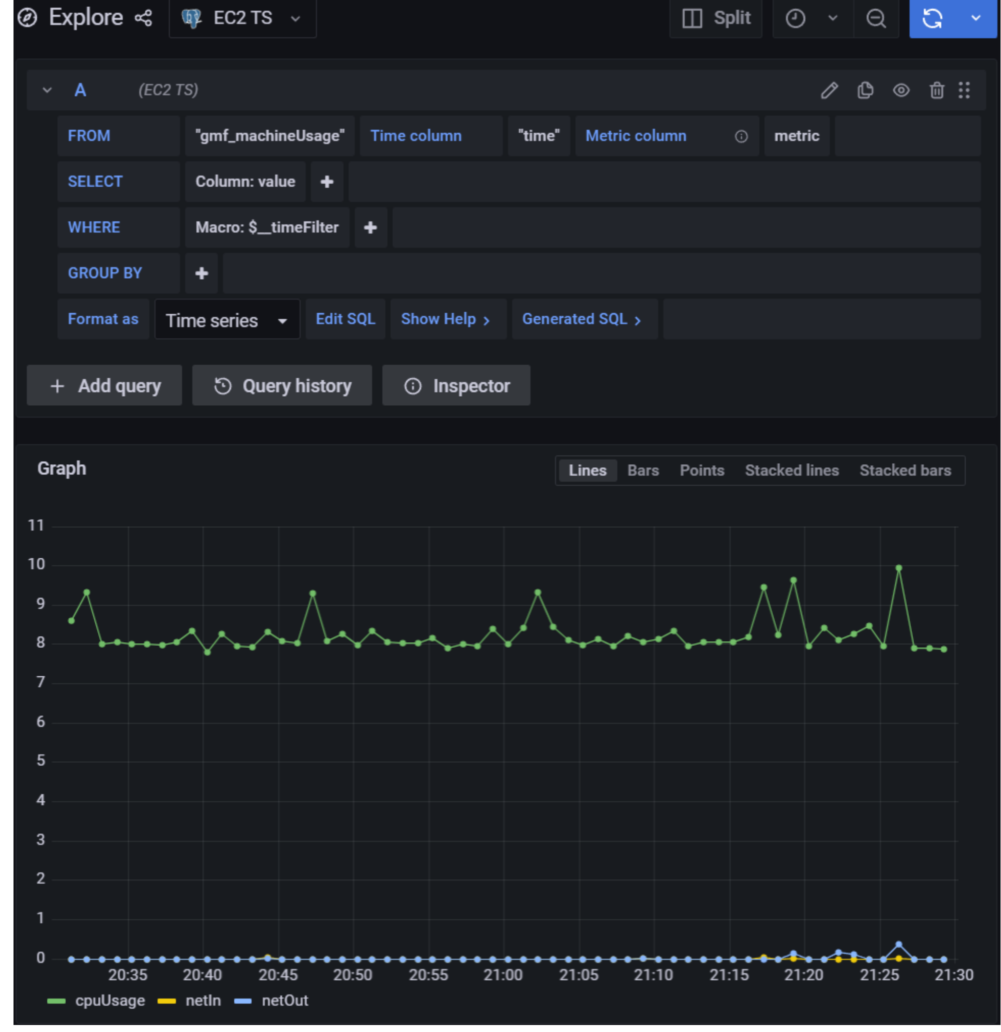
This step requires contact with a Grafana administrator. They will create a Grafana account with a temporary password and grant you access to the database instance. Once your account has been created, visit https://vis.uvasif.org/ and sign in using the username and password provided to you. Change your password immediately for account security. In the bottom left corner hover over your profile icon and select "Preferences". Scroll down to the "Organizations" section, and verify your role as a LivingLinkLab editor is your current role. It should look like this:



Once you have selected the LivingLinkLab Editor role, visit the "Explore" page via the button on the left navigation tray, shown off to the right on this page. On the "Explore" page, verify that the data source is set to "EC2 TS" at the top. Select the app you wish to visualize data from by typing your app into the box displaying "user_demoApp". The format is "username_appName"; if my username was "user" and the app I wanted to use was "demoApp", then the "user_demoApp" shown in the image below would be correct.



Although not required to make the visualization work, it is generally a good idea to change the metric column from "none" to "metric". This will greatly improve the visualization for apps containing multiple metrics, but can also improve single-metric apps by labeling the time series with the appropriate metric name. Here is an example of a visualization generated through this simple interface:



Don't forget to set the time window in the upper right-hand corner of the page using the clock icon!

V. Using the API

The SIF website and Grafana will cover most of the simple use cases users may encounter. For more advanced operations or for certain "power users", it could be nice to interact with the SIF platform programmatically. For this, we grant all users access to the SIF API immediately upon account creation. There are two base API URLs: one for bulk data downloads and one for everything else. In this document, we provide examples on how to use one endpoint from each of these API URLs. Full documentation is available at these links:

Regular API: https://api.uvasif.org/v2/documentation/

Download API: https://download.uvasif.org/v1/documentation/

Note: All endpoints, aside from the documentation ones, are secured using a token-bearer authentication scheme. For these endpoints, you will need to provide a valid Cognito-issued JWT in the "Authorization" header of the request.

a) Example #1: Retrieve a list of your apps.
   i) According to the documentation, the endpoint we should use is **`/v2/apps/list`**. This is part of the regular API, so the full URL we need to use is https://api.uvasif.org/v2/apps/list.
   ii) Command:
```
$ curl  -header "Authorization: yourJwtHere"
        https://api.uvasif.org/v2/apps/list
```
   iii) API response:
```
{
      "code": 200,
      "apps": [
            "myApp",
            "demoApp",
            "SIF-test"
      ]
}
```
   iv) Done! The API should always return a JSON according to the schemas described in the documentation. If an invalid token is provided, the API will respond with a HTTP 401 (Unauthorized) error.

b) Example #2: Download data from an app.
   i) According to the documentation, the endpoint we should use to download all metrics from an app is **`/v1/<app_name>/all`**. In this example, we will download data from the "demoApp" app between July 1st, 2021 at 13:00:00 EST and November 15th 2021 at 20:30:00 EST. This is a part of the download API, so the full URL we need to use is https://download.uvasif.org/v1/demoApp/all.
   ii) The download endpoints can make use of some additional query string arguments. We can use these arguments to provide the desired data range we defined above.

The start time is **1625158800.000** in Unix epoch time (seconds). Similarly, the end time is **1637026200.000**. These are added to the request URL in the next step.

iii) Command:

```
$ curl  -header "Authorization: yourJwtHere"
        https://download.uvasif.org/v1/demoApp/all?start=1625158800.000&end=1637026200.000
```

iv) API response: a csv containing the data from the given app in the given time range. If no data is available, the CSV will only have a single row containing the column headers.

v) Done! The API should always return either a CSV with data (0+ rows) or a JSON that describes some error message. More information regarding the specifics of usage and the various responses can be found in the API documentation, linked above.

VI. That's it!

If you were able to reach this point, you're all set up and ready to fully enjoy the benefits of the SIF platform. Reach out to a SIF administrator if you encounter issues.