Miro Holopainen

Delivering energy consumptions using Azure messaging

Metropolia University of Applied Sciences

Master of Engineering

Information Technology

Master's Thesis

31 October 2021

PREFACE

This is for you my little boy. Your father is waiting for you to born and wishes you a fare welcome to the world. This is also for you my dear pregnant wife, who feeds me when I'm hungry and keeps my sugar levels at maximum by providing constantly candy while writing this thesis.

And you Henri. Thank you for dropping in from time to time to distract me.

Seriously though, thank you Martti for constantly pushing the new ideas and challenging the existing ones as I went.

Akseli, thank you too for weeding out my bad ideas and planting good ideas in my head instead.

And Antti. Thank you for inspiring me just when I needed it the most.

Klaukkala, 31 October 2021
Miro Holopainen

Abstract

Author:	Miro Holopainen
Title:	Delivering energy consumptions using Azure messaging
Number of Pages:	59 pages
Date:	31 October 2021
Degree:	Master of Engineering
Degree Programme:	Information Technology
Professional Major:	Software Development
Supervisors:	Sami Sainio, Principal Lecturer

This thesis was an assignment from a company called EG EnerKey. EnerKey had a dated energy consumption delivery and storing system that needed a replacement. The old system could no longer fulfil all the requirements EnerKey has today. To solve the problem a code-based solution, which can deliver energy consumptions from their source to the final database, was created. It can be used by other components all while utilizing Microsoft Azure cloud offerings and Influx time series database. One of the desired outcomes was also to generally just create something with different Azure offering, so that EnerKey will have more knowledge on them.

Thesis provided a background of what needed to be done and also a study on different Azure messaging offerings and how good would they fit for EnerKey and the replacement product. Technical implementation was done utilizing three different messaging products and creating a couple C# based libraries utilizing those products, and an Azure function, that can handle the energy consumption data delivery to the database.

Thesis resulted in a solution that can be used to replace the EnerKey's current system while achieving most of the requirements set for the solution. Thesis also provided recommendation on the single Azure messaging product. Solution was not fully completed as the time run out, but it serves as an excellent basis to continue replacing the old system. Thesis result can be used to deliver other kinds of data using Azure too, though few adjustments for the end product must be made on that case.

Keywords: Azure, Enterprise messaging, Influx, Time series, C#

Contents

List of Abbreviations

C#	Programming language created by Microsoft
CLI	Command line interface
DI	Dependency injection
EMS	Enterprise messaging system
FIFO	First-in, first-out
FQDN	Fully qualified domain name
JSON	JavaScript object notation, a serialization format
MaaS	Messaging as a service
ORM	Object-relational mapping
Protobuf	Google Protocol Buffer, a serialization format
SaaS	Software as a service
SLA	Service level agreement

1 Introduction

Handling time series data in modern ways within cloud, can be a daunting task. Time series data can take all forms and shapes. It is generated constantly and most of the time, the amount of timeseries data tends to grow. While working with EG EnerKey, their time series data originates mostly from energy consumptions. This energy consumption can be electricity (e.g., How much electricity is consumed by lights), water (e.g., how many litres are consumed within the building) to district heating (e.g., how much a building is heated). Within EnerKey, there are over 100 different types of these. Most of the time, the energy consumption is read from the meters, that are spread out to EnerKey's customers buildings.

There are over 50 000 facilities (as EnerKey call them) with over 100 000 metering points that delivers, or EnerKey get consumption from, automatically. With raw numbers, this means that within a single month, there is need to process ~72 million datapoints. Most of these are read at least once a day, meaning that the system currently must handle over 2 million datapoints a day. This system is also required to scale, as EnerKey is looking at an explosive growth within Nordic countries, meaning that the system should handle at least double the numbers here in the future.

While there are lots of ways to handle this, this thesis focuses on the cloud native approach. EnerKey is using Microsoft Azure cloud platform, where almost everything behind their SaaS product is hosted. So, utilizing on premise servers and services is out of the scope and only Azure-based services can be used on the final implementation. EnerKey has existing dual database solution based on InfluxDB, that has grown to a non scalable state. The current solution is unnecessary complex and does not pass EnerKey's quality standards for the written software anymore. It has couple design flaws and does not scale properly anymore. EnerKey had to do some workarounds to counter these points. In this thesis work, ideal result would be a product that can replace the

whole dual database solution and bring a simpler and scalable solution to its place.

Delivering energy consumptions is also one of the most important parts within the EnerKey SaaS as most of their business models are built on top of the measured energy consumptions. It has to be secure, readily available and fault tolerant. Most of the customers want to access their energy consumption as fast as possible, so solution should also be a performant.

Addition to this, EnerKey has need to test out other messaging offerings Azure have. While EnerKey is already utilizing Azure Service bus, it should not be the only Azure messaging product used. In the end a recommendation is given for the most appropriate technology.

This thesis provides background on selected mechanisms to transfer and store time series data, using Microsoft Azure. Thesis also describes the system as it should be implemented and what to expect, watch out and other relevant notes of the system. Finally, there are conclusions describing how did this whole process was done and what was learned along the way.

2 Time series data

As stated by E. Clower [1], data that is collected at different points in time, is a time series data. She also states that [1] time series data is opposed to cross-sectional data, which is observations in a single point in time. She continues to state [1] that as data points originate from adjacent time periods, there is possibility of correlation between the observations, which distinguishes cross-sectional data from time series data.

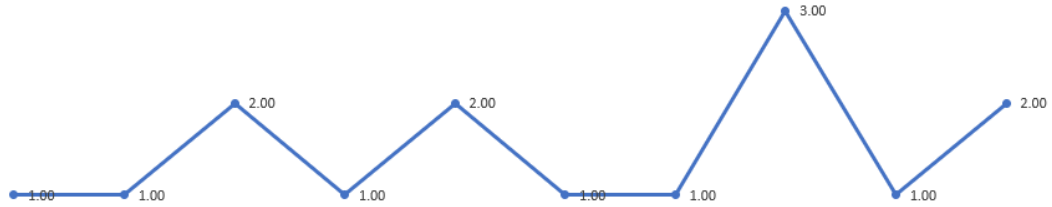

Figure 1. Observations in regular time interval from excel.

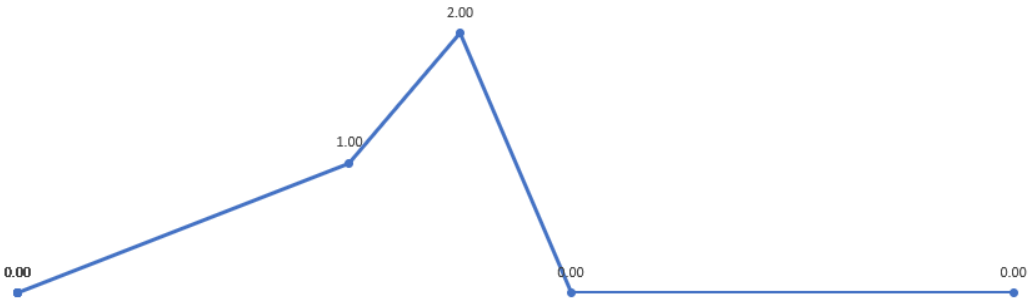

Figure 2. Observations in irregular time interval from excel.

In InfluxData [2], they state that time series data can be classified into two types: Observations at regular or irregular time intervals as presented in figures 1 and 2. They also state [2] that the time series data is different from regular data with key difference that you ask questions about it over the time. If one of the axes you are using is time, you are using time series.

In EnerKey, one of the key time series data is energy consumption, meaning that how much energy was consumed within a timeframe. For example: a retail store used 500 kwh worth of electricity yesterday. They can also use it to compare previous values: an office used 150% less water than this same date a year ago. When multiple same or different quantities are collected, EnerKey can use it to create a comprehensive understanding on the whole picture about the energy consumed and how it has changed over time. This time series data is collected in two different shapes.

2.1 Interval Consumption

Interval consumption is a consumption measured at regular intervals. This regularity can come in few different intervals, but the interval always stays the same, as long as the same meter is used. For example, an hourly consumption. Hourly consumption is always reported as an hourly interval and it can be read once an hour or in longer periods. The value is an energy consumed within that single point in time.

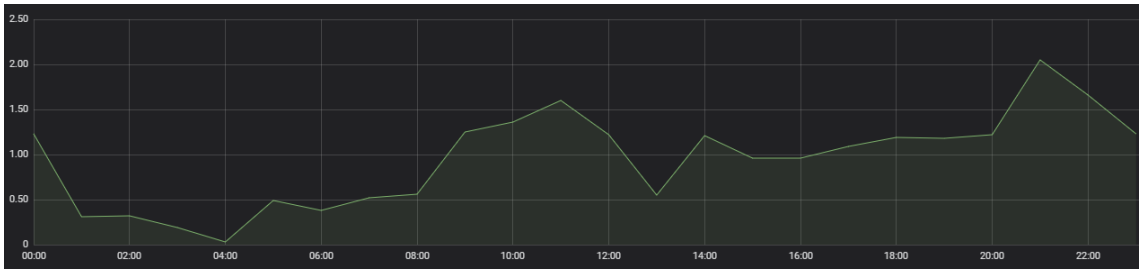

Figure 3. Hourly measured energy consumption as seen in EnerKey.

As seen in the chart 3, energy was measured from yesterday, and it contains 24 datapoints. Every hour timestamp has value for energy. EnerKey must be able to collect these intervals:

- Less than 15 minutes
- 15 minutes
- 1 hour
- 1 month

2.2 Cumulative consumption

Cumulative consumption is a consumption measured at irregular intervals. This means that each time consumption is measured, it reports a cumulative reading of consumed energy after the previous measurement. The measured value is always at least the same or bigger, as long as the same meter is used. For example, a meter is read once an hour. Previous hour value was 10 and this hour value is 20, meaning that overall, 10 units of energy has been consumed within 1 hour.

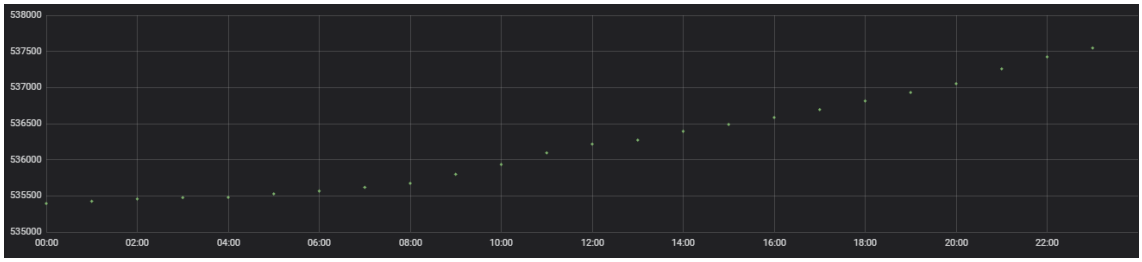

Figure 4. Cumulative measured energy consumption as seen in EnerKey.

As seen in the figure 4, a cumulative consumption is growing each hour. In EnerKey this must be converted to an interval consumption at some point, as almost all the reporting in the EnerKey SaaS is based on interval consumptions alone, not cumulative.

2.3 Other types of energy consumptions

Interval consumption and cumulative consumption is basis of all energy reporting within EnerKey SaaS. It does not stop there, as there are more energy-based concepts too. For example, a peak power.

As noted by Sunpower-uk [3] a maximum power that is sustained for a short period of time, is called peak power. Transferring and reporting these are also needed, but they are out of the scope of this thesis.

3 Enterprise integration

In this time series data scenario, EnerKey require creation, transfer and storing mechanism for the system. For satisfying the set of required features for the implementation, a way of handling the integration between the meters and the usage of their data, must be decided.

Hohpe et al. [4] define enterprise integration as a task to make disparate applications for producing unified set of functionalities, while typically running in multi-computer and multi-platform system. They also state that the approach to integration can be summed up to these integration styles [4]:

- File transfer
- Shared database
- Remote procedure invocation
- Messaging

Hohpe et al. [4] define that file transfer and shared database enable sharing the data, but not the functionality. Alternative is a remote procedure invocation. But as they say [4], making remote calls tend to be slower and more failure oriented. To reduce rate of failure, they [4] suggest that data transfer should be done in an asynchronous way, so that sender does not require waiting of a receiver.

This will lead to a last scenario: messaging.

Messaging is process of sending small amount of information frequently so that it allows different applications to collaborate behaviourally and share data at the same time. When doing messaging asynchronously, there is no need for both systems to be running at the same time. Messages are delivered through channel that connect sender and a receiver. [4.]

3.1 Machine to machine messaging

Messaging is the approach that was chosen to integrate energy consumptions from meters to usable time series data for EnerKey. This will allow them to create a loosely coupled system, which exchange information about the energy consumptions using a set of messages in a way that is still performant.

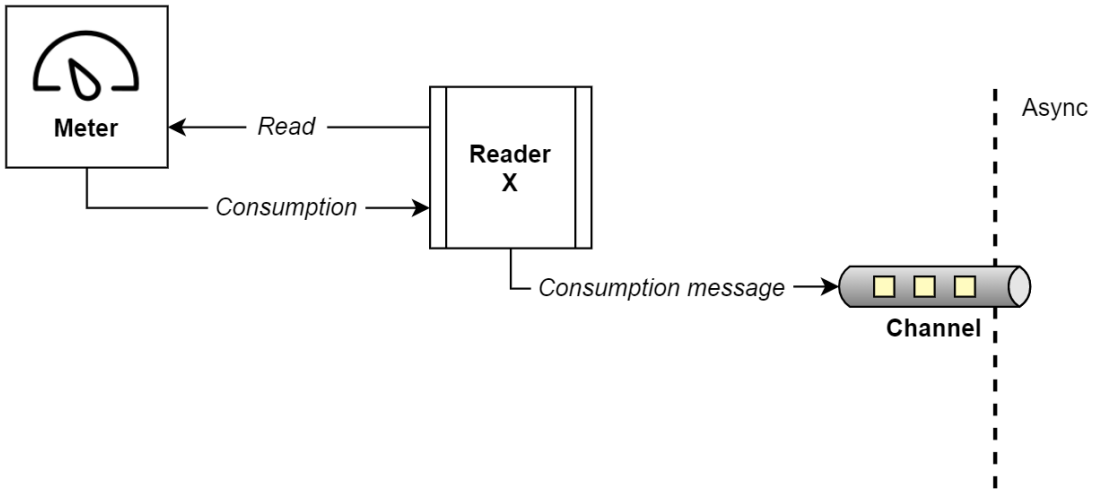

Figure 5. Getting consumption and forwarding it as a message.

As seen from the figure 5, when the consumption reader is invoked, it reads the consumptions from the meter and then creates a message and sends it forward to channel.

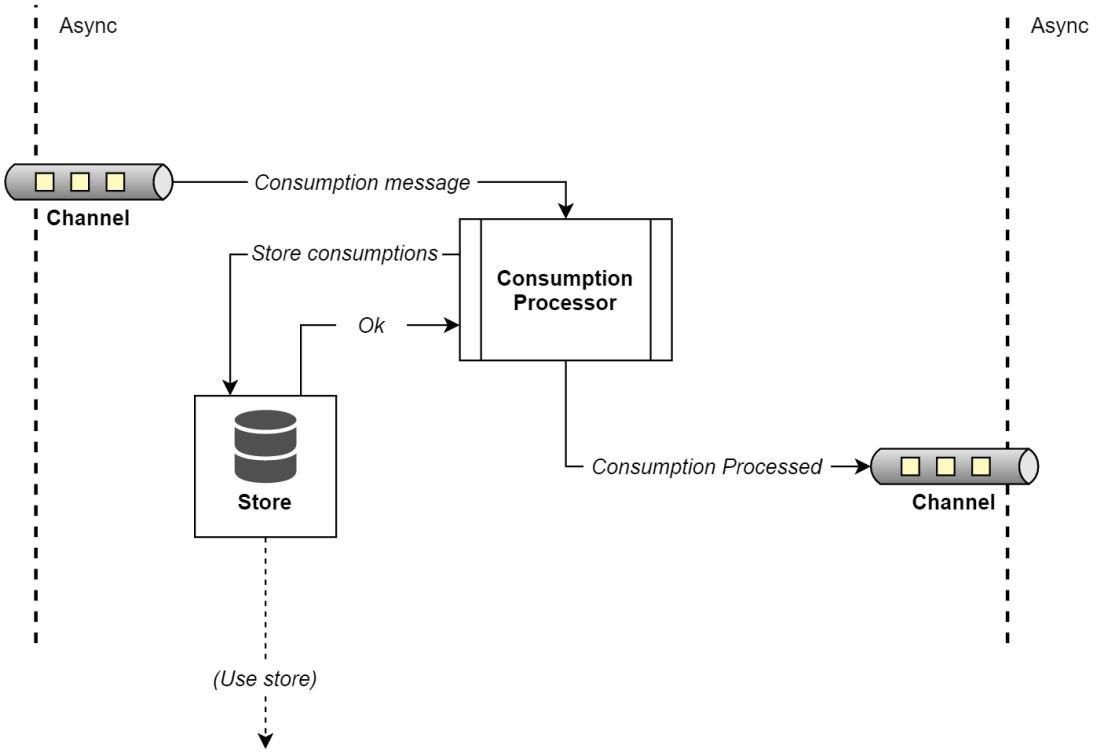

Figure 6. Storing time series.

As seen from the figure 6, consumption processor receives the consumption message, transforms it and stores it into a separate store. It then sends a consumption processed message forward to channel, where it can be used for extra processes that should be done after time series has been saved. Meanwhile store is usable and serves time series data for the components asking it.

3.2 Azure solutions for messaging

EnerKey SaaS is hosted in Azure, so messaging solution should be a native Azure offering. As seen in Azure message services product page, these products are available for handling this type of scenario [5].

3.2.1 Azure Relay

Azure relay is meant for securely exposing services running within on-premise network [6]. It is not applicable for this scenario, as no on-premise component is involved.

3.2.2 Azure SignalR Service

SignalR service allows real-time communication between web applications [7]. It is not applicable for this scenario, as no real time communication is required.

3.2.3 HDInsight

HDInsight is a way to run some popular open-source frameworks within Azure. Apache Kafka is a distribution platform, that could be used [8]. While it could potentially handle this scenario, it was not selected due to the fact that EnerKey has zero experience with it, while there are other Azure services that has been used successfully in other types of scenarios before within EnerKey.

3.2.4 Notification Hubs

Notification hub is a notification platform, meant for pushing notifications to the mobile devices [9]. It is not applicable for this scenario as no mobile devices are involved.

3.2.5 Azure IoT Hub

Azure IoT Hub is a platform to create a secure and reliable way to communicate with IoT application and IoT devices [10]. Due to the small 4 KB message size [11], handling 2.4 million datapoints a day would most likely raise a pricing to highest tier, becoming a most expensive solution. Therefor it was dropped for this scenario, but EnerKey could potentially benefit from the Azure IoT Hub on other types of scenarios where only small set of data needs to be transferred.

3.2.6 Azure Web PubSub

Azure Web PubSub is real time messaging platform for developing web applications using native and serverless WebSocket's [12]. It is in preview phase. It is not applicable for this scenario as no real time communication is required and it is not production ready yet.

3.2.7 Azure Event Grid

Azure Event Grid is event-based routing mechanism build on top of Http-protocol. It utilizes publisher subscriber model. It does support a CloudEvents 1.0 standard and is available with Kubernetes. Event Grid could trigger a serverless function to push data forward. [13.]

As Event Grid does not have any undesired features, this product was selected for implementation.

3.2.8 Azure Event hub

Event hub is built to ingest real-time data and then stream events and allows building of dynamic data pipelines. It has geo-replication features with geo-disaster scenarios. Event hub supports AMQP, HTTPS and Apache Kafka as its communication layer. [14.]

As Event Hub does not have any undesired features, this product was selected for implementation.

3.2.9 Azure Service Bus

Azure Service Bus is highly reliable messaging service, providing cloud based MaaS. It is a message brokering service allowing asynchronous operations with FIFO pattern using queue mechanism and publisher/subscriber pattern with topics and subscription mechanism. [15.]

As Event Grid does not have any undesired features and it has been utilized within EnerKey before, this product was selected for implementation.

3.2.10 Queue Storage

Build on top of HTTP or HTTPS protocols, Queue Storage provides a store for large number of messages, allowing access from anywhere. It does not state reliability and supports FIFO pattern only. [16.]

As FIFO pattern is the only one supported, Queue Storage has risk of not satisfying all the requirements and due to the time constraints of thesis, this product was not selected for further implementation, but it has other potential use cases for EnerKey.

3.3 Serialization

Serialization is a way to make an object state to fit into a format that is transmissible and storable. [17.]

Due to the fact that messages might have a maximum message size and high performance is required, one has to look about a way to serialize messages properly. While there are many ways to handle serialization, EnerKey already utilizes JSON for general payloads, and Google Protobuf for high performance, low memory scenarios, so only these two are further analysed.

3.3.1 JSON

JSON is part of the subset in JavaScript language. JSON has no additional features JavaScript does not already have. JSON is not a programming language but it is a data interchange format. Data within a JSON is collection of name and value pairs or an ordered list of values. This makes it a human readable format. [18.]

3.3.2 Google Protobuf

Protobuf is a Google's message serialization technique for a language and platform neutral way to serialize a structured data to be used with communication protocol, data storage and more. It is non-human readable, efficient binary format, where a message can be serialized and deserialized using .proto description of the message. [19.]

With C#, EnerKey is using third party library, that creates .proto internally using C# annotations and therefor no additional files are required for it to work in their scenario.

3.4 JSON and Protobuf comparison in C#

When comparing JSON and Protobuf and their possible usage with C#, one must note that there are multiple different code libraries for each. Therefor, comparison between the two, is done using the most popular EnerKey serialization libraries: Microsoft's System.Text.Json for JSON and protobuf-net for Protobuf.

Base of testing is an example consumption timeseries payload. Testing is done with .NET5.0 version, using 24-hour period for timeseries, as it is most common payload in EnerKey.

```
namespace JsonProtobufComparer
{
    // Install-Package protobuf-net --Version 3.0.101
    // Install-Package BenchmarkDotNet --Version 0.13.1
    using BenchmarkDotNet.Running;

    public class Program
    {
        public static void Main()
        {
            BenchmarkRunner.Run(typeof(Program).Assembly);
        }
    }
}
```

Listing 1. Benchmark runner.

```
namespace JsonProtobufComparer
{
    using System;
    using System.Collections.Generic;
    using ProtoBuf;

    [Serializable]
    [ProtoContract]
    public class Payload
    {
        [ProtoMember(1)]
        public int MeterId { get; set; }
            = 123456;
        [ProtoMember(2)]
        public int Kind { get; set; }
            = 0;
        [ProtoMember(3)]
        public IReadOnlyCollection<Datapoint> Datapoints
            => GenerateTimeSeries();

        private static IReadOnlyCollection<Datapoint>
            GenerateTimeSeries()
        {
            const int points = 24;
            var result = new List<Datapoint>(24);
            var startingPoint =
                new DateTime(2021, 1, 1, 0, 0, 0, DateTimeKind.Utc);
            for (var i = 0; i < points; i++)
                result.Add(new Datapoint
                {
                    Timestamp = startingPoint
                        .Add(TimeSpan.FromHours(i)),
                    Value = 123456789
                });
            return result;
        }
    }

    [Serializable]
    [ProtoContract]
    public class Datapoint
    {
        [ProtoMember(100)]
        public DateTime Timestamp { get; set; }
        [ProtoMember(101)]
        public double Value { get; set; }
    }
}
```

Listing 2. Test data.

Using the code described in listing 1 and 2, a program can start running different benchmarks which measure CPU time and memory allocation.

3.4.1 Serialization benchmark

Next test is measuring performance when object is transformed to its wire format: string for JSON and byte-array for protobuf.

```
namespace JsonProtobufComparer
{
    using System.IO;
    using System.Text.Json;
    using BenchmarkDotNet.Attributes;
    using ProtoBuf;

    [MemoryDiagnoser]
    public class SerializationTest
    {
        public static Payload Payload =>
            new();

        [Benchmark]
        public string SerializeWithJson()
            => JsonSerializationTest.Serialize(Payload);

        [Benchmark(Baseline = true)]
        public byte[] SerializeWithProtobuf()
            => ProtoSerializationTest.Serialize(Payload);
    }

    public static class JsonSerializationTest
    {
        public static string Serialize(Payload p)
        {
            return JsonSerializer.Serialize(p);
        }
    }

    public static class ProtoSerializationTest
    {
        public static byte[] Serialize(Payload p)
        {
            using var ms = new MemoryStream();
            Serializer.Serialize(ms, p);
            ms.Position = 0;
            return ms.ToArray();
        }
    }
}
```

Listing 3. Serialization benchmarks

When the benchmark as described in listing 3 has been run, it results in the following results:

```
BenchmarkDotNet=v0.13.1, OS=Windows 10.0.19043.1237 (21H1/May2021Update)
AMD Ryzen 9 5900X, 1 CPU, 24 logical and 12 physical cores
.NET SDK=6.0.100-rc.1.21463.6
  [Host]     : .NET 5.0.10 (5.0.1021.41214), X64 RyuJIT
  DefaultJob : .NET 5.0.10 (5.0.1021.41214), X64 RyuJIT


|               Method |     Mean |     Error |    StdDev | Ratio | RatioSD |  Gen 0 | Allocated |
|--------------------- |---------:|----------:|----------:|------:|--------:|-------:|----------:|
|     SerializeWithJson | 7.263 us | 0.0344 us | 0.0287 us |  2.34 |    0.02 | 0.2518 |      4 KB |
| SerializeWithProtobuf | 3.109 us | 0.0250 us | 0.0222 us |  1.00 |    0.00 | 0.1297 |      2 KB |
```

Figure 7. Serialization benchmark results.

As can be seen from the figure 7, when serializing JSON with C#, it is 2.34 times slower than protobuf, while allocating two times more memory.

3.4.2 Deserialization benchmark

Next test is about deserializing test data generated from listing 2, using serialized content from listing 3.

```
namespace JsonProtobufComparer
{
    using System;
    using System.Text.Json;
    using BenchmarkDotNet.Attributes;
    using ProtoBuf;

    [MemoryDiagnoser]
    public class DeserializationTest
    {
        public static string SerializedJson =>
            JsonSerializationTest.Serialize(new());

        public static byte[] SerializedProto =>
            ProtoSerializationTest.Serialize(new());

        [Benchmark]
        public Payload DeserializeWithJson()
            => JsonDeserializationTest.Deserialize(SerializedJson);

        [Benchmark(Baseline = true)]
        public Payload DeserializeWithProtobuf()
            => ProtoDeserializationTest.Deserialize(SerializedProto);
    }

    public static class JsonDeserializationTest
    {
        public static Payload Deserialize(string json)
        {
            return JsonSerializer.Deserialize<Payload>(json);
        }
    }

    public static class ProtoDeserializationTest
    {
        public static Payload Deserialize(ReadOnlyMemory<byte> proto)
        {
            return Serializer.Deserialize<Payload>(proto);
        }
    }
}
```

Listing 4. Deserialization benchmarks.

```
BenchmarkDotNet=v0.13.1, OS=Windows 10.0.19043.1237 (21H1/May2021Update)
AMD Ryzen 9 5900X, 1 CPU, 24 logical and 12 physical cores
.NET SDK=6.0.100-rc.1.21463.6
  [Host]     : .NET 5.0.10 (5.0.1021.41214), X64 RyuJIT
  DefaultJob : .NET 5.0.10 (5.0.1021.41214), X64 RyuJIT


|                  Method |      Mean |     Error |    StdDev | Ratio | RatioSD |  Gen 0 | Allocated |
|------------------------ |----------:|----------:|----------:|------:|--------:|-------:|----------:|
|     DeserializeWithJson | 10.262 us | 0.0260 us | 0.0243 us |  1.74 |    0.02 | 0.2441 |      4 KB |
| DeserializeWithProtobuf |  5.889 us | 0.0629 us | 0.0589 us |  1.00 |    0.00 | 0.2594 |      4 KB |
```

Figure 8. Deserialization benchmark results.

When deserialization back to C# object, JSON is 1.74 times slower, while consuming same amount of memory.

3.4.3 Longer timeseries period benchmark

Last test is about making a timeseries period a longer. EnerKey is required to read whole meter history in special cases and it can contain up to 10 years of hourly data. So, the next test is using 87648 points as a period, instead of 24.

```
BenchmarkDotNet=v0.13.1, OS=Windows 10.0.19043.1237 (21H1/May2021Update)
AMD Ryzen 9 5900X, 1 CPU, 24 logical and 12 physical cores
.NET SDK=6.0.100-rc.1.21463.6
  [Host]     : .NET 5.0.10 (5.0.1021.41214), X64 RyuJIT
  DefaultJob : .NET 5.0.10 (5.0.1021.41214), X64 RyuJIT


|                  Method |     Mean |    Error |   StdDev | Ratio | RatioSD |     Gen 0 |     Gen 1 |     Gen 2 | Allocated |
|------------------------ |---------:|---------:|---------:|------:|--------:|----------:|----------:|----------:|----------:|
|     SerializeWithJson | 31.91 ms | 0.388 ms | 0.363 ms |  2.43 |    0.05 | 625.0000 | 593.7500 | 468.7500 |     27 MB |
| SerializeWithProtobuf | 13.08 ms | 0.262 ms | 0.331 ms |  1.00 |    0.00 | 734.3750 | 718.7500 | 562.5000 |      9 MB |
```

Figure 9. Serialization benchmark result for 10 years of data.

```
BenchmarkDotNet=v0.13.1, OS=Windows 10.0.19043.1237 (21H1/May2021Update)
AMD Ryzen 9 5900X, 1 CPU, 24 logical and 12 physical cores
.NET SDK=6.0.100-rc.1.21463.6
  [Host]     : .NET 5.0.10 (5.0.1021.41214), X64 RyuJIT
  DefaultJob : .NET 5.0.10 (5.0.1021.41214), X64 RyuJIT


|                  Method |     Mean |    Error |   StdDev | Ratio | RatioSD |     Gen 0 |     Gen 1 |     Gen 2 | Allocated |
|------------------------ |---------:|---------:|---------:|------:|--------:|----------:|----------:|----------:|----------:|
|     DeserializeWithJson | 44.08 ms | 0.865 ms | 1.094 ms |  1.66 |    0.07 | 1272.7273 | 1181.8182 | 1181.8182 |     31 MB |
| DeserializeWithProtobuf | 26.57 ms | 0.516 ms | 0.756 ms |  1.00 |    0.00 | 1500.0000 | 1437.5000 | 1000.0000 |     16 MB |
```

Figure 10. Deserialization benchmark results for 10 years of data.

As can be seen from figure 9, with longer period, JSON is 2.43 time slower while allocating three times the memory. Then it can be seen from the figure 10, deserialization is 1.66 times slower with JSON while allocating almost double the amount of memory. When accounting all these benchmarks, it is clear that

the protobuf is a winner, gaining much more performance over JSON, while losing ability to be human readable.

It is surely beneficial to able to decipher consumptions messages on anywhere. Being able to open protobuf message within the wire, a deserialization step is required for protobuf, as it is binary format. This is a more of a corner case, where most of the time, no human has to be involved in between the consumption message users. When it is needed, it is more for a troubleshooting type of scenario, that does not occur daily for EnerKey.

A protobuf was selected as a transformation- and store format for the consumption messages.

3.5 Storage

As seen in the chapter 3.4.3, a timeseries message can in some cases be in megabytes alone. When comparing size limitations of selected azure products, it results in a following limitation [20, 21, 22]:

- Azure Service bus has size limitation for message of 256 KB – 1 MB based on pricing tier.
- Azure Event hub has size limitation of publication of 256 KB – 1 MB based on pricing tier.
- Azure Event Grid has size limit for event of 1 MB.

In the most common scenarios, limits from any of those products are okay, as they won’t be exceeded. When reading a complete history, for example 10 years, limits are easily exceeded.

In this case, message must be split to multiple messages, so that all will fit the limitations. While working independently scaling and non-FIFO message handlers, splitting a message will lead to a potential problem in their ordering while creating more traffic in the process. Other alternative would be to not allow long history reads, but it is not an option for EnerKey. Therefor a new method to handle messages should be introduced.

If message's timeseries component (which makes up most of the size of the message) can be stored in the separate store, then only a metadata message needs to be delivered. A location of the actual timeseries payload is then stored to metadata message. Now, when this metadata message is handled, the handler can fetch that timeseries data from this separate store and continue processing.

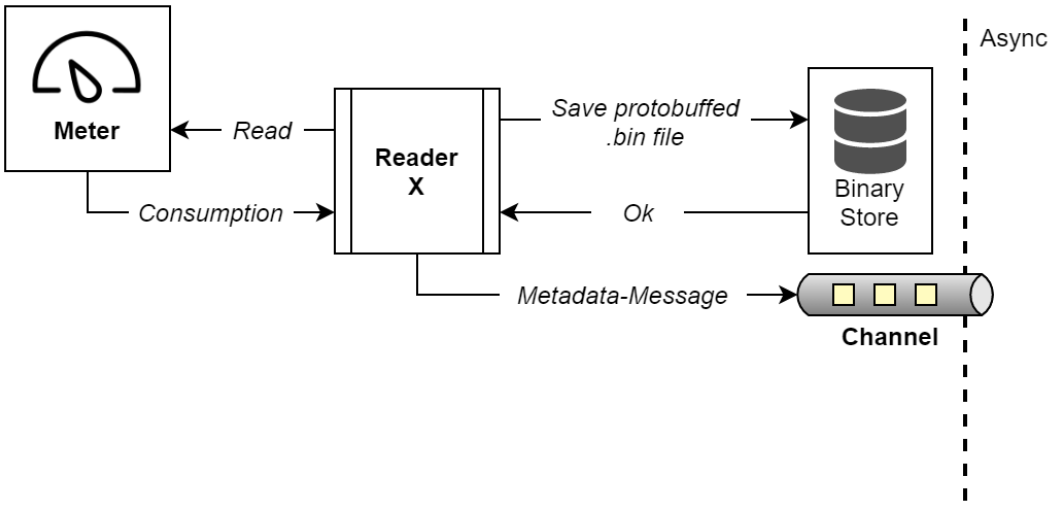

Figure 11. Timeseries payload saved to binary store.

As seen from the figure 11, instead of sending whole payload forward to a channel, the actual timeseries is saved to the binary store and only metadata message is sent forward to the channel. This allows message to be small as possible and the actual timeseries payload is stored for later processing. This way (almost) an unlimited amount of data can be read in single operation and still utilize messaging. The payload is transformed to Protobuffed binary file and saved to some store that can handle files.

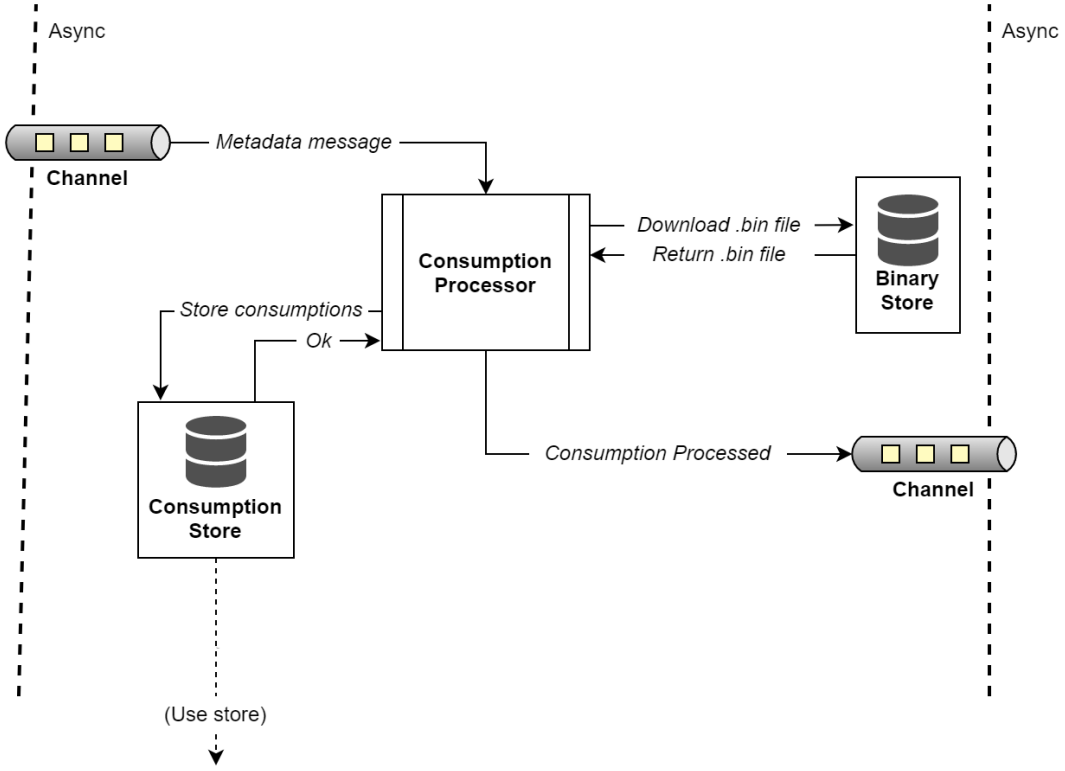

Figure 12. Using binary store.

Now later, when processing the metadata message, file location can be read from the message and then download it to the processor as seen in figure 12. As it is in Protobuffed binary format, it needs to be opened with the original proto definition and then it can be deserialized and stored to the consumption store.

3.5.1 Azure Blob Storage

Azure Blob Storage is secure and scalable storage for unstructured data at massive scale. Azure Blob Storage is supported by .NET and other popular languages, acting as a cornerstone to any serverless architecture. [23.]

In EnerKey's scenario, Azure Blob Storage is a best fit for their needs. They can use it as a binary store for all the timeseries payloads, that are in the Protobuffed format. When saving a blob or data to the storage, they can utilize properties and add their own metadata to the blob.

There are two different kinds of properties and metadata available: System properties and User-defined metadata. With C#, system properties are maintained by the Azure Storage client library, while user-defined metadata is collection of name-value pairs, that are completely fillable by the user. [24.]

As there are over 100 000 meters sending a data every day, so there will be massive number of binary files. For EnerKey this means that they can add different kind of information, such as meter id, to the blob. When troubleshooting or accessing blobs later, it can be done easily by utilizing filters on these properties. Having massive number of binary files, also means that each day the amount of space required, keeps growing. This will incur a cost increase every day, which is not desired. To solve this, EnerKey can utilize cost optimizations provided by the Azure.

Azure Blob Storage costs depend on the access tier of the blob. Blobs can be in the following tiers: Premium, Hot, Cool and Archive. They differ on availability, durability, retrieval latency and throughput characteristics and redundancy configurations. [25.]

Azure blob Storage provides way to automatically manage the lifetime of the blob. Lifecycle management policies are set through collection of JSON rules in the storage. [26.]

With this scale, manually managing the tiers is not an option. Setting up lifecycle management policies, EnerKey can transition blobs from Hot tier to Cool tier and after a while, to an archive tier and finally delete the whole blob. There is no clear requirement set for this, so adjustments to the lifetimes are out of the scope of this thesis and EnerKey must realigned them later.

3.5.2 InfluxDB

When selecting database for storing the energy consumptions, it is beneficial to select a time series database. These kinds of databases are optimized for timeseries. There are many to choose from, few notable examples: InfluxDB,

Prometheus, Kdb+ and Apache Druid. In EnerKey, they have utilized InfluxDB for couple of years and have expertise with it, so it is the primary pick for the time series database.

InfluxDB is purpose-built time series database that supports collection, storing, monitoring, visualisation and alerting of time series data. InfluxDB data model utilizes name of the measurement, set of tags, set of fields and a timestamp. [27.]

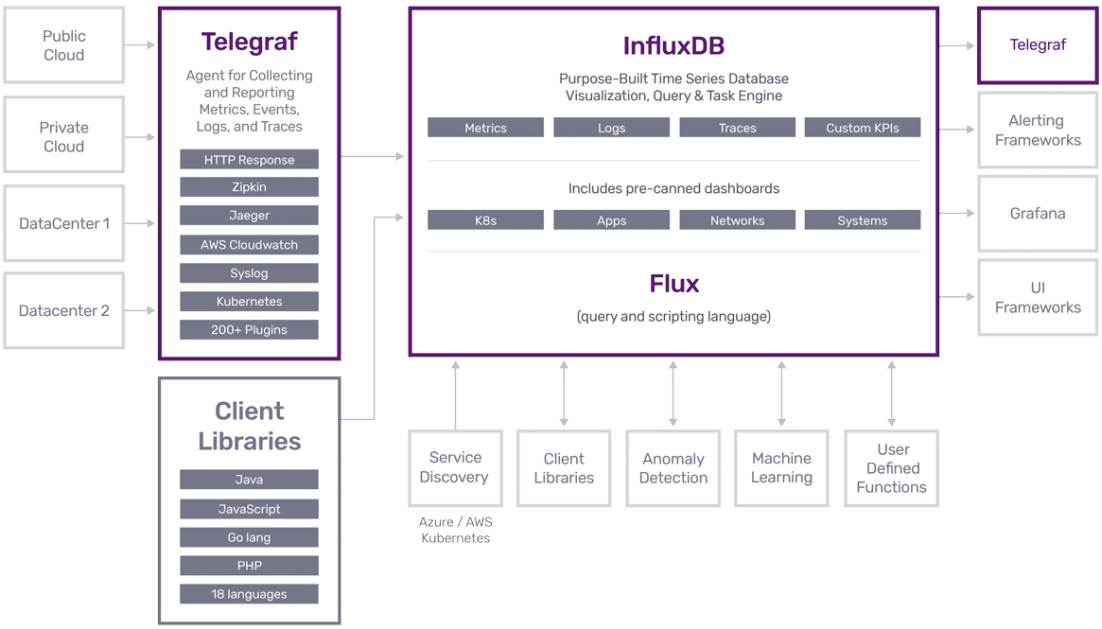

Figure 13. Influx stack and different components [28].

As seen from figure 13, data is written to InfluxDB using either Telegraf or client libraries provided by the InfluxData. EnerKey uses C# client libraries to write the data to the InfluxDB, using their self-built schema for energy consumptions. Later the data is queried from the InfluxDB using the Flux query language. As EnerKey utilizes service and repository patterns, the official InfluxData provided client library was a bit too broad and therefor requires a separate layer to achieve this. This custom-built abstraction layer is described later in the chapter 5.1.

3.6 Event handling

Every time a data is read from the meter, it must be transported and possibly transformed. So, an event is a series of timestamped data and contains necessary information to process it as described in the chapters 3.1 and 3.5.1. This event must be handled in the cloud. There are couple Azure options for this: Azure Functions, Kubernetes and Logic apps. EnerKey has not utilized Kubernetes yet. Logic apps are GUI driven setup and EnerKey prefer infrastructure as a code, so Azure Functions is the choice for event handling.

3.6.1 Azure Functions

Azure Functions is event-driven serverless computing platform to handle orchestration problems. Functions can be built and debugged locally and then deployed to the cloud. Functions integrate to services with triggers and bindings. Functions can be written with wide variety of programming languages. There are templates available for Visual Studio and Functions integrate with CI/CD pipeline in the Azure Pipelines. [29.]

Consumption processor, as described in the chapter 3.5.1, will therefor have a trigger for a message. After the message is triggered, it is binded to a message format. Then the handling can start, where the timeseries payload is fetched from the Azure Storage. Lastly the processor saves a formed timeseries to the InfluxDB and sends a new message forward for other EnerKey post processes.

4 Product implementation

This chapter describes an implementation built on previously selected components. Implementation utilizes three different channels (Azure Event Grid, Azure Event Hub and Azure Service bus) and recommendation should be given based on their fit for EnerKey in this scenario. In some implementation cases, it contains parts of EnerKey’s trade secrets, so only interface level implementation is given in these cases. There will be code behind those interfaces in the final product, it just isn’t available within this thesis.

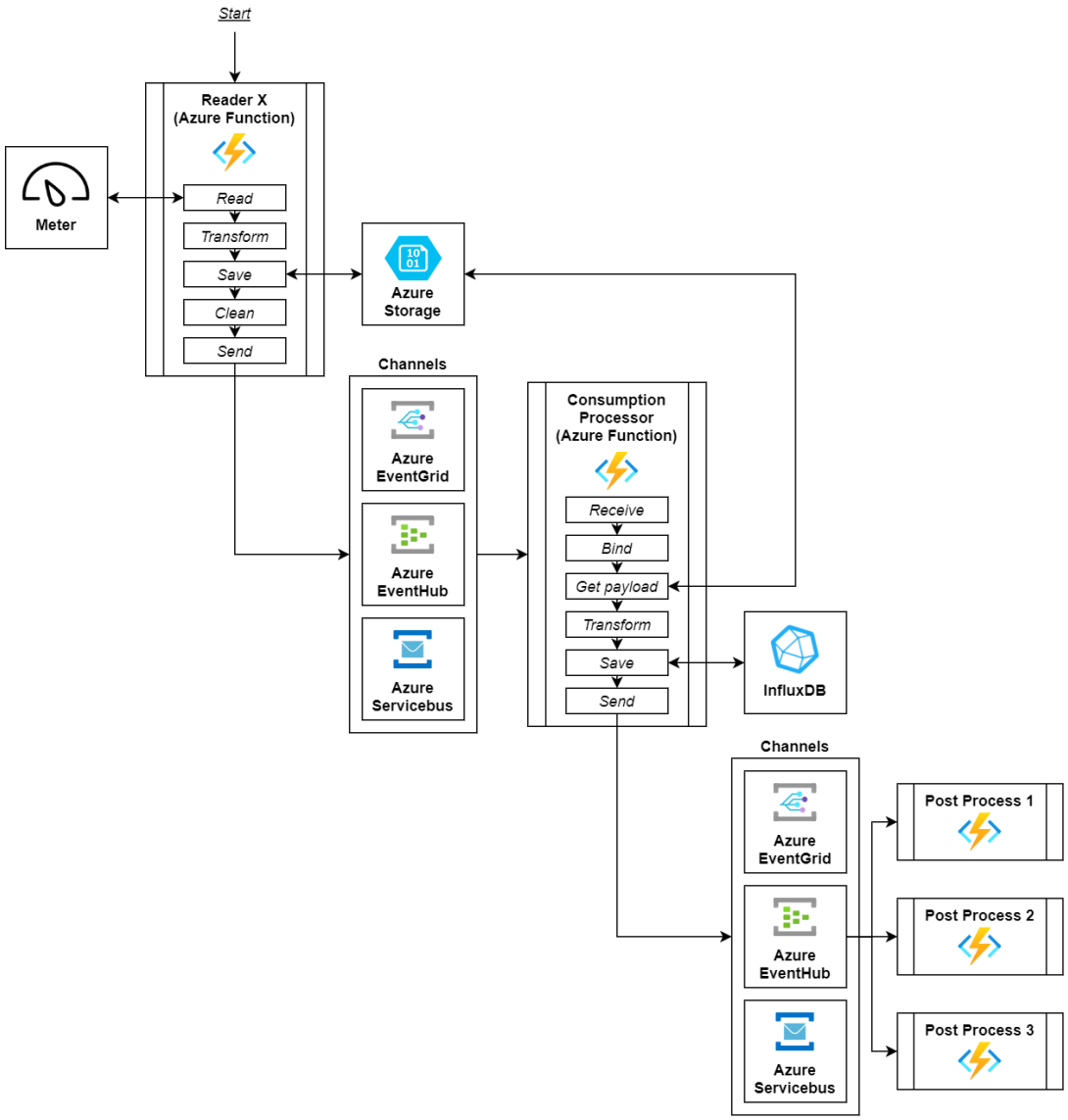

Figure 14. Components of the implementation.

As can be seen from the figure 14, a reader is Azure function, that will first read the actual timeseries data from the meter, then transform it to generic consumption payload. This payload will be then saved to Azure Storage and the location of the saved item is picked up. Resources get cleaned and a message of meters metadata and consumption metadata is formed. Message is sent forward to channels, where it will be picked up by the Consumption Processor. It will bind the message and use it to fetch the payload from the Azure Storage. Consumption payload will be then transformed to a generic timeseries and will be saved to InfluxDB. When process is done, a new consumption saved

message is sent forward to next level of channels, where they are available for post processes EnerKey has.

4.1 Setting up Azure Resources

When setting up Azure resources there is couple options: through Azure Portal, with Azure CLI or Azure SDK packages. Setup was done with Azure CLI. EnerKey Reader X is existing Azure Function and has its own setup outside of this implementation, so it is not included here.

First thing that must be done with Azure CLI, is to login. Then Id of the subscription must be noted as the resources will be created there.

```
az login
az account list
```

Listing 5. Azure CLI login and account selection.

Next, select id the subscription and create a new resource group. these are defined as variables, as the information will be required later.

```
SubId="<fill-id>"
RG="thesis-resources"
az account set --subscription $SubId
az group create --location westeurope --resource-group $RG
```

Listing 6. Set account and create resource group.

Now there is resource group where all the other resources will be created. Next step is to create a storage account that can store the payload binary files.

```
StrAcc="consumptions"
az storage account create \
     --name $StrAcc \
     --location westeurope \
     --resource-group $RG \
     --sku Standard_LRS \
     --kind StorageV2 \
     --access-tier Hot \
     --allow-shared-key-access true
```

Listing 7. Creation of Azure Storage that will contain binary files.

Now the storage account has been created, next the Azure Function for processing consumption message is created.

```
FuncName="consumption-functions"
az functionapp create --name $FuncName \
    --resource-group $RG \
     --storage-account $StrAcc \
     --consumption-plan-location westeurope \
     --os-type Windows
```

Listing 8. Creation of Azure Function application.

Next, Azure Event Hub is created.

```
HubNamespace="hub-consumption-ns"
Hub="hub-consumptions"
az eventhubs namespace create \
     --name $HubNamespace \
     --resource-group $RG \
     --location westeurope
az eventhubs eventhub create \
     --name $Hub \
     --resource-group $RG \
     --namespace-name $HubNamespace
```

Listing 9. Creation of Event Hub and its namespace.

Next, Azure Service bus, topic and subscription will be created.

```
Sb="bus-consumptions-ns"
SbTopic="consumptions-topic"
SbSub="consumptions-sub"
az servicebus namespace create \
     --name $Sb \
     --resource-group $RG \
     --location westeurope \
     --sku Standard
az servicebus topic create \
     --name $SbTopic \
     --namespace-name $Sb \
     --resource-group $RG
az servicebus topic subscription create \
     --name $SbSub \
     --namespace-name $Sb \
--topic-name $SbTopic \
     --resource-group $RG
```

Listing 10. Creation of Service bus and its topic and subscription.

Setting up the event grid is last thing that needs to be done.

```
GridDomain="grid-consumptions-domain"
GridTopic="grid-consumption-topic"
Sub="grid-consumption-sub"
az eventgrid domain create \
      --name $GridDomain \
      --resource-group $RG \
      --identity systemassigned \
      --input-schema eventgridschema \
      --location westeurope
```

Listing 11. Creation of Event Grid with EventGridSchema.

Now all the required components are created. Event grid requires an additional step to add subscription where the events are routed to. It can be done once the Azure Function that receives the event is created and deployed to Azure.

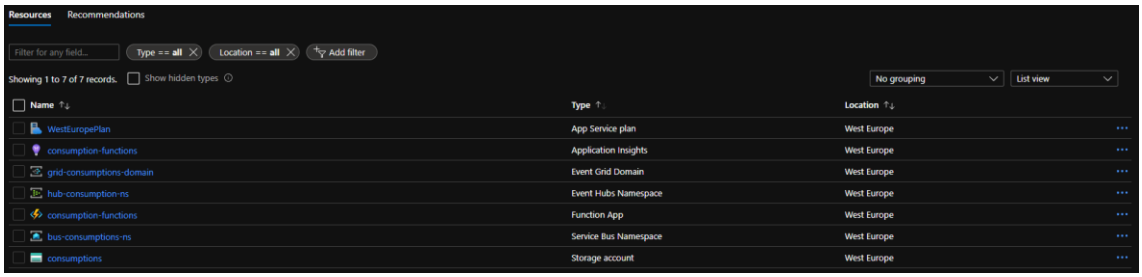

Figure 15. Completed setup with all resources in Azure.

Azure setup now looks like in figure 15.

4.2 EnerKey.Api.Messaging

For creating a message that contains timeseries payload, other methods to convert it to the protobuf and open it from the protobuf, another abstraction layer is required. It is named as EnerKey.Api.Messaging.

This will be a .NET5.0 library, packed as a NuGet package, that can be shared organization wide from EnerKey's private NuGet feed. Library should contain classes for message and other services to save payload as a blob and to send message to selected channels.

A new .NET5.0 Library project is created and relevant packages are added from Nuget.org:

- Install-Package Azure.Messaging.EventHubs
- Install-Package Azure.Messaging.ServiceBus
- Install-Package Azure.Messaging.EventGrid
- Install-Package Azure.Storage.Blobs
- Install-Package protobuf-net

In the next chapter, the message is defined as a class.

4.2.1 Consumption message

This message contains all the metadata EnerKey requires and the actual timeseries containing collection of timestamp value pairs. Message class also implements save location for the binary file in Azure storage, and various other methods to make a message usage neat and clean.

Message utilizes protobuf with the help of protobuf-net library for C#. With this library, .proto files are not required, and everything can be handled using attribute decorations: [ProtoContract] and [Protomember(n)]. [30.]

```
namespace EnerKey.Api.Messaging.Standard.Consumption.V1
{
    using System;
    using System.Collections.Generic;
    using System.IO;
    using Azure.Messaging.EventGrid;
    using Azure.Messaging.EventHubs;
    using Azure.Messaging.ServiceBus;
    using ProtoBuf;

    [Serializable]
    [ProtoContract]
    public class ConsumptionMessage
    {
      ...
    }
}
```

Listing 12. Consumption message.

Consumption message is a class with distinct namespace for consumption and version. Whenever breaking changes are introduced later, a new version namespace should be created. This will allow EnerKey to handle these breaking changes with the messages. The public class ConsumptionMessage is decorated with protobuf-net attributes.

```
[ProtoMember(1)]
public int MeterId { get; set; }

[ProtoMember(2)]
public HashSet<Timeseries> Series { get; set; }
     = new();

[ProtoMember(3)]
public string WriteLocation { get; set; }

[ProtoMember(4)]
public string TransactionId { get; set; }

// Other metadata fields omitted
// ...
```

Listing 13. Consumption message properties.

Next, all the public properties are added and decorated with protobuf-net Protomember attributes. Protomember requires a tag that indicates field location within the Protobuffed payload. HashSet is used for timeseries that the timeseries key value pairs are truly unique within the collection.

```
#region Serializers
/// <summary>
///     To Protobuffed binary
/// </summary>
/// <returns></returns>
private byte[] ProtobufInstance()
{
     using var ms = new MemoryStream();
     Serializer.Serialize(ms, this);
     ms.Position = 0;
     return ms.ToArray();
}

/// <summary>
///     From Protobuffed binary
/// </summary>
/// <param name="protobufBytes"></param>
/// <returns></returns>
public static ConsumptionMessage Open(
     ReadOnlyMemory<byte> protobufBytes)
{
     return Serializer
          .Deserialize<ConsumptionMessage>(protobufBytes);
}
#endregion
```

Listing 14. Protobuf serialization and deserialization.

Next, couple methods are implemented to make serialization to and from protobuf, an easy task. Creating a binary array, it should happen only within the message itself, therefor private access is used. This method allows EnerKey to create a protobuf version of the current instance of the message. Whenever the

binary array is later required to be changed back to message, a library user can just call ConsumptionMessage.Open() and give the binary array as a parameter and the deserialized instance will be returned.

```
#region Helpers for various information for the message itself
[ProtoIgnore] private const string Subject
     = nameof(ConsumptionMessage);

[ProtoIgnore] private const string ContentType
     = "application/protobuf";

/// <summary>
///     Get current version
/// </summary>
/// <returns></returns>
public static string Namespace()
{
     return typeof(ConsumptionMessage).Namespace;
}

/// <summary>
///     Check if received namespace matches current version
/// </summary>
/// <param name="receivedNamespace"></param>
/// <returns></returns>
public static bool CanOpen(string receivedNamespace)
{
     return receivedNamespace == Namespace();
}

/// <summary>
///     Get .bin filename
/// </summary>
/// <returns></returns>
public string FileName()
{
     return $"meter-{MeterId}/" +
             $"{MeterId}_{TransactionId}.bin";
}
#endregion
```

Listing 15. Helper methods for consumption message.

Next, few things to describe the message itself are prepared, such as its name (Subject) and in what content type it is in. Also, what version (Namespace) library is using, and if it can be actually opened to the same version.

```
#region Message wrappers to create transport object
/// <summary>
///     Create <see cref="EventData" /> from this message
/// </summary>
/// <returns></returns>
/// <exception cref="ArgumentNullException"></exception>
public EventData EventHubEnvelope()
{
     if (string.IsNullOrWhiteSpace(WriteLocation))
          throw new ArgumentNullException(WriteLocation);

     return new EventData(ProtobufInstance())
     {
          Properties =
          {
               {"EventId", $"{MeterId}-{TransactionId}"},
               {"Subject", Subject},
               {"ContentType", ContentType},
               {"Namespace", Namespace()}
          }
     };
}

/// <summary>
///     Create <see cref="ServiceBusMessage" /> from this message
/// </summary>
/// <returns></returns>
/// <exception cref="ArgumentNullException"></exception>
public ServiceBusMessage ServiceBusEnvelope()
{
     if (string.IsNullOrWhiteSpace(WriteLocation))
          throw new ArgumentNullException(WriteLocation);

     return new ServiceBusMessage(
          new BinaryData(ProtobufInstance()))
     {
          MessageId = $"{MeterId}-{TransactionId}",
          ContentType = ContentType,
          Subject = Subject,
          ApplicationProperties =
          {
               {"Namespace", Namespace()}
          }
     };
}

/// <summary>
///     Create <see cref="EventGridEvent"/> from this message
/// </summary>
/// <returns></returns>
public EventGridEvent EventGridEnvelope()
{
     return new EventGridEvent(
          Subject,
          "custom",
          Namespace(),
          ProtobufInstance())
     {
          Id = $"{MeterId}-{TransactionId}"
     };
}
#endregion
```

Listing 16. Transport object wrappers.

Last thing that is needed, is to use these self-descriptions mentioned before and create a relevant transport object that the Azure Services can use: EventData for Event Hub, ServiceBusMessage for Service bus and EventGridEvent for Event Grid as seen from listing 16.

4.2.2 Timeseries implementation

Next thing is to create a class for the actual timeseries, which is timestamp value pair. It is used as a hash set as seen in the previous chapter.

```
namespace EnerKey.Api.Messaging.Standard.Consumption.V1
{
    using System;
    using ProtoBuf;

    [Serializable]
    [ProtoContract]
    public class Timeseries : IEquatable<Timeseries>
    {
        public Timeseries(DateTime timestamp)
        {
            Timestamp = timestamp;
        }

        [ProtoMember(1)]
        public DateTime Timestamp { get; }

        [ProtoMember(2)]
        public double Value { get; set; }

        #region Equatable

        public override int GetHashCode()
            => Timestamp.GetHashCode();

        public bool Equals(Timeseries other)
            => Timestamp == other?.Timestamp;

        public override bool Equals(object obj)
            => obj is Timeseries converted
               && Timestamp == converted.Timestamp;

        #endregion
    }
}
```

Listing 17. Timeseries class.

Here protobuf-net is imported again, as the timeseries class will be Protobuffed too. It also implements IEquatable, as timeseries is used as a hash set in the message and this enforces uniqueness within a set.

4.2.3 Event grid implementation

Next, sending an event grid event to Azure service created in chapter 4.1, is implemented.

```
namespace EnerKey.Api.Messaging.EventGrid
{
    using System;
    using System.Collections.Generic;
    using System.Net;
    using System.Threading.Tasks;
    using Azure;
    using Azure.Messaging.EventGrid;
    using Microsoft.Extensions.Options;

    public interface IEventGridClient
    {
        Task SendEventGridEventsAsync(
            IReadOnlyList<EventGridEvent> events);
    }

    public class EventGridClient : IEventGridClient
    {
        private readonly EventGridPublisherClient _eventGrid;
        public EventGridClient(IOptions<EventGridOptions> options)
        {
            _eventGrid = new EventGridPublisherClient(
                new Uri(options.Value.Url),
                new AzureKeyCredential(options.Value.AccessKey));
        }

        public async Task SendEventGridEventsAsync(
            IReadOnlyList<EventGridEvent> events)
        {
            var sendResult = await _eventGrid
                .SendEventsAsync(events);
            if (sendResult.Status is not (int) HttpStatusCode.OK)
                throw new InvalidOperationException(
                    "Eventgrid sending failed");
        }
    }

    public class EventGridOptions
    {
        public string Url { get; set; }
        public string AccessKey { get; set; }
    }
}
```

Listing 18. IEventGridClient and implementation.

Here Azure.Messaging.EventGrid library is used. It contains EventGridPublisherClient. EventGridPublisherClient requires an authentication against Azure, so that is done using IOptions pattern containing a relevant EventGridOptions. This client is then wrapped within EnerKey's own abstraction layer. IEventGridClient is an interface that can be injected using DI where ever

sending a list of Event Grid events is required. The implementation sends the events and throws if the status is not HttpStatusCode.OK.

4.2.4 Event hub implementation

Next, sending an event hub event to Azure service created in chapter 4.1, is implemented.

```
namespace EnerKey.Api.Messaging.EventHub
{
    using System;
    using System.Collections.Generic;
    using System.Threading;
    using System.Threading.Tasks;
    using Azure.Messaging.EventHubs;
    using Azure.Messaging.EventHubs.Producer;
    using Microsoft.Extensions.Options;

    public interface IEventHubClient
    {
        Task SendEventHubEventsAsync(
            IReadOnlyList<EventData> events,
            CancellationToken cancellationToken = default);
    }

    public class EventHubClient : IEventHubClient, IAsyncDisposable
    {
        private readonly EventHubProducerClient _eventHub;

        public EventHubClient(IOptions<EventHubOptions> options)
        {
            _eventHub = new EventHubProducerClient(
                options.Value.ConnectionString,
                options.Value.HubName);
        }

        public async Task SendEventHubEventsAsync(
            IReadOnlyList<EventData> events,
            CancellationToken cancellationToken = default)
        {
            using var eventBatch = await _eventHub
                .CreateBatchAsync(cancellationToken);

            foreach (var eventData in events)
                eventBatch.TryAdd(eventData);

            await _eventHub.SendAsync(
                eventBatch,
                cancellationToken);
        }

        public async ValueTask DisposeAsync()
        {
            await DisposeAsyncCore();
            GC.SuppressFinalize(this);
        }

        protected virtual async ValueTask DisposeAsyncCore()
        {
            await _eventHub
                .DisposeAsync()
                .ConfigureAwait(false);
        }
    }

    public class EventHubOptions
    {
        public string ConnectionString { get; set; }
        public EventHubProducerClientOptions HubName { get; set; }
    }
}
```

Listing 19. IEventHubClient and its implementation.

Here Azure.Messaging.EventHubs library is used. It contains EventHubProducerClient. Using IOptions pattern, authentication can be setup for it. It implements IAsyncDisposable to clean the client on dispose. IEventHubClient is EnerKey's abstraction layer that can be injected using DI where ever event grid sending is required. Actual implementation creates an event batch and then sends this batch to Event hub in Azure.

4.2.5 Service bus implementation

Last messaging related implementation is Service Bus.

```
namespace EnerKey.Api.Messaging.Servicebus
{
    using System;
    using System.Collections.Generic;
    using System.Threading;
    using System.Threading.Tasks;
    using Azure.Messaging.ServiceBus;
    using Microsoft.Extensions.Options;

    public interface IEnerkeySbClient
    {
        Task SendServiceBusMessagesAsync(
            IReadOnlyList<ServiceBusMessage> messages,
            CancellationToken cancellationToken = default);
    }

    public class EnerkeySbClient : IEnerkeySbClient, IAsyncDisposable
    {
        private readonly ServiceBusClient _serviceBusClient;
        private readonly string _topic;

        public EnerkeySbClient(IOptions<ServiceBusOptions> options)
        {
            _serviceBusClient = new ServiceBusClient(
                options.Value.ConnectionString);

            _topic = options.Value.TopicName;
        }

        public async Task SendServiceBusMessagesAsync(
            IReadOnlyList<ServiceBusMessage> messages,
            CancellationToken cancellationToken = default)
        {
            await using var sender = _serviceBusClient
                .CreateSender(_topic);

            await sender
                .SendMessagesAsync(messages, cancellationToken);
        }

        public async ValueTask DisposeAsync()
        {
            await DisposeAsyncCore();
            GC.SuppressFinalize(this);
        }

        protected virtual async ValueTask DisposeAsyncCore()
        {
            await _serviceBusClient
                .DisposeAsync()
                .ConfigureAwait(false);
        }
    }

    public class ServiceBusOptions
    {
        public string ConnectionString { get; set; }
        public string TopicName { get; set; }
    }
}
```

Listing 20. IEnerkeySbClient and its implementation.

Here Azure.Messaging.ServiceBus library is used. It contains ServiceBusClient. Using IOptions pattern, authentication is setup for it. For cleaning the resources,

IAsyncDisposable is implemented. IEnerKeySbClient abstraction is created, and it can be injected using DI where ever Service bus message sending is required. Actual implementation creates a SenderClient for topic defined in ServicebusOptions and then sends those messages forward.

4.2.6 Azure storage implementation

Last thing that needs to be implemented, is blob handling.

```
namespace EnerKey.Api.Messaging.Blob
{
    using System;
    using System.Collections.Generic;
    using System.IO;
    using System.Linq;
    using System.Threading;
    using System.Threading.Tasks;
    using Azure.Storage;
    using Azure.Storage.Blobs;
    using Microsoft.Extensions.Options;
    using ProtoBuf;
    using Standard.Consumption.V1;

    public interface IBlobClient
    {
        Task<byte[]> DownloadBlobAsync(
            Uri writeLocationUri,
            CancellationToken cancellationToken = default);

        Task UploadBlobs(
            IReadOnlyList<ConsumptionMessage> messages,
            CancellationToken cancellationToken = default);
    }
    ...
}
```

Listing 21. IBlobClient interface.

This interface is abstraction on top of Azure.Storage.Blobs library. It contains a download method to get .bin file from the Azure Storage and another method to upload to blobs from messages.

```
public class BlobClient : IBlobClient
{
      private readonly BlobServiceClient _blobServiceClient;
      private readonly AzureStorageOptions _options;
      private readonly StorageTransferOptions _storageTransferOptions;

      public BlobClient(IOptions<AzureStorageOptions> options)
      {
            _options = options.Value;

            _blobServiceClient = new BlobServiceClient(
                  _options.ConnectionString);

            _storageTransferOptions = new StorageTransferOptions
            {
                  MaximumConcurrency = _options.MaxConcurrency,
                  MaximumTransferSize = _options.TransferSize
            };
      }
      ...
}
```

Listing 22. BlobClient implementation constructor.

IOptions pattern is used again to setup an BlobServiceClient, AzureStorageOptions and StorageTransferOptions.

```
public async Task<byte[]> DownloadBlobAsync(
      Uri writeLocationUri,
      CancellationToken cancellationToken = default)
{
      var blobResource = new BlobResourceUrl(writeLocationUri);

      var containerClient = _blobServiceClient
            .GetBlobContainerClient(blobResource.Container);
      var consumptionMessageBlob = containerClient.GetBlobClient(
            blobResource.FilePath);

      await using var ms = new MemoryStream();
      await consumptionMessageBlob
            .DownloadToAsync(ms, cancellationToken);

      ms.Position = 0;
      return ms.ToArray();
}
```

Listing 23. DownloadBlobAsync method.

Downloading a blob is pretty straight forward. Message metadata contains a write location, so when it is given a BlobResourceUrl class is utilized to define the container and actual path where the file will be in Azure Storage. Container client is created for the resource, and then the blob is downloaded into a memory stream. It is then returned as array of bytes. These are the Protobuffed bytes from the .bin file.

```
private static IDictionary<string, string>
GetPayloadMetadata(ConsumptionMessage payload)
{
      var tNamespaceStringArr = typeof(ConsumptionMessage)
            .Namespace
            ?.Split(".") ?? Array.Empty<string>();

      var version = tNamespaceStringArr[^1];
      return new Dictionary<string, string>
      {
            {"serializer", "protobuf"},
            {"version", $"{version}".ToLower()},
            {"meterid", payload.MeterId.ToString()},
            {"transactionid", payload.TransactionId}
      };
}

private async Task<IReadOnlyList<BlobContainerClient>> GetBlobContainersAsync(
      int payloadCount,
      CancellationToken cancellationToken = default)
{
      var result = new List<BlobContainerClient>(payloadCount);

      var containerName = _options.ConsumptionContainer.ToLower();

      var createContainerClient =
            _blobServiceClient.GetBlobContainerClient(containerName);

      await createContainerClient.CreateIfNotExistsAsync(
            cancellationToken: cancellationToken);

      for (var i = 0; i < payloadCount; i++)

      result.Add(_blobServiceClient.GetBlobContainerClient(containerName));

      return result;
}
```

Listing 24. BlobContainerClient and metadata methods.

Uploading blobs are a bit more complicated. First, GetPayloadMetadata() method is needed. It can decorate an actual blob with metadata. Metadata decoration allows inspection later when blob is viewed. GetBlobContainersAsync() allows every blob to have its own container client.

```
public async Task UploadBlobs(
      IReadOnlyList<ConsumptionMessage> messages,
      CancellationToken cancellationToken = default)
{
      var blobContainers =
            await GetBlobContainersAsync(messages.Count, cancellationToken);

      for (var i = 0; i < messages.Count; i++)
      {
            var payload = messages[i];
            var container = blobContainers[i];

            var file = payload.FileName();
            var blobClient = container.GetBlobClient(file);

            await using (var ms = new MemoryStream())
            {
                  Serializer.Serialize(ms, payload);
                  ms.Position = 0;

                  await blobClient.UploadAsync(
                        ms,
                        metadata: GetPayloadMetadata(payload),
                        transferOptions: _storageTransferOptions,
                        cancellationToken: cancellationToken);
            }

            payload.WriteLocation =
                  $"{_blobServiceClient.Uri}{container.Name}/{file}";
      }
}
```

Listing 25. Blob uploading.

Last thing then is to implement UploadBlobs() method, that will actually handle the blob file upload. File is serialized to Protobuf and added to memory stream. Then the memory stream is uploaded to the Azure Storage. After the write completes, a write location is noted and added for the message. Write location now contains the path for the .bin file within Azure Storage.

This will conclude the EnerKey.Api.Messaging implementation. It is now ready to be used as a library, and contains all relevant implementations for messaging and Azure Storage handling.

4.3 Azure functions

Next step is to handle message within Azure Functions. In this chapter, two different types of Azure functions are described.

4.3.1 EnerKey Readers

EnerKey has over 80 different integrations to different energy- and meter platforms. Timeseries data is read from these different sources and transformed to a unified standard. In this standard implementation, timeseries should be unified and then create the messages. Before exiting, messages should be sent forward.

```
public abstract class ReaderBase
{
      private readonly IBlobClient _blobClient;
      private readonly IEventGridClient _eventGridClient;
      private readonly IEventHubClient _eventHubClient;
      private readonly IEnerkeySbClient _enerkeySbClient;
      protected ReaderBase(
            IBlobClient blobClient,
            IEventGridClient eventGridClient,
            IEventHubClient eventHubClient,
            IEnerkeySbClient enerkeySbClient)
      {
            _blobClient = blobClient;
            _eventGridClient = eventGridClient;
            _eventHubClient = eventHubClient;
            _enerkeySbClient = enerkeySbClient;
      }

      // Everything else omitted as this class considered as an trade secret
      // A abstract method should call SendMessagesAsync() before exit.

      private async Task SendMessagesAsync(
            IReadOnlyList<ConsumptionMessage> messages,
            CancellationToken cancellationToken = default)
      {
            var eventHubEvents =
            new List<EventData>(messages.Count);
            var eventGridMessages =
            new List<EventGridEvent>(messages.Count);
            var serviceBusMessages =
            new List<ServiceBusMessage>(messages.Count);

            await _blobClient.UploadBlobs(messages, cancellationToken);

            // Blob payload and drop data points to optimize wire size
            foreach (var msg in messages)
                  msg.Series = new HashSet<Timeseries>(0);

            // Envelope
            eventHubEvents.AddRange(
                  messages.Select(payload => payload.EventHubEnvelope()));
            serviceBusMessages.AddRange(
                  messages.Select(payload => payload.ServiceBusEnvelope()));
            eventGridMessages.AddRange(
                  messages.Select(payload => payload.EventGridEnvelope()));

            // Send
            await _eventHubClient
                  .SendEventHubEventsAsync(eventHubEvents, cancellationToken);
            await _enerkeySbClient
                  .SendServiceBusMessagesAsync(serviceBusMessages,
cancellationToken);
            await _eventGridClient
                  .SendEventGridEventsAsync(eventGridMessages);
      }
}
```

Listing 26. Sending messages from the EnerKey readers.

All different readers should use this SendMessagesAsync() method, that will upload message to Azure Storage and then envelope to corresponding message and sent with matching client implemented in previous chapter.

4.3.2 Consumption Processor

This Consumption Processor is the Azure Function that contains triggers for the corresponding messages. For this a new .NET5.0 Azure Function project is created. It is named to EnerKey.Consumption.Functions. Next, setup of the three different triggers is implemented.

```
namespace EnerKey.Consumption.Functions
{
    public class ServiceBusTrigger
    {
        private readonly IBlobClient _blobClient;
        private readonly IConsumptionService _consumptionService;

        public ServiceBusTrigger(
            IConsumptionService consumptionService,
            IBlobClient blobClient)
        {
            _consumptionService = consumptionService;
            _blobClient = blobClient;
        }

        [Function(nameof(ServiceBusTrigger))]
        public async Task HandleServiceBusMessages(
            [ServiceBusTrigger(
                "consumptions-topic",
                "consumptions-sub",
                Connection = "ServiceBusConnection",
                IsBatched = true)]
            byte[][] messages,
            FunctionContext executionContext)
        {
            var payloadSplit = messages
                .Select(m => ConsumptionMessage.Open(m))
                .ToList();

            await SavePayloadsAsync(payloadSplit);
        }

        private async Task SavePayloadsAsync(
            IReadOnlyList<ConsumptionMessage> payloads)
        {
            foreach (var payload in payloads)
            {
                var blobBytes = await _blobClient
                    .DownloadBlobAsync(new Uri(payload.WriteLocation));

                var blobPayload = ConsumptionMessage
                    .Open(blobBytes);

                await _consumptionService
                    .CreateAndSaveConsumptionAsync(blobPayload);
            }
        }
    }
}
```

Listing 27. Processing Service bus messages in Azure Function.

ServiceBusTrigger, that receives messages as a batch of byte arrays, is implemented. These byte arrays are Protobuffed messages. ConsumptionMessage helper method created in chapter 4.2.1, called Open(), is then used. Then all these messages are looped. Injected IBlobClient is used to fetch the messages from the Azure Storage. For this EnerKey can utilize the message metadata property called WriteLocation. When the single message has been fetched and opened, it is passed to a IConsumptionService.

```
namespace EnerKey.Consumption.Functions
{
    public class EventHubTrigger
    {
        private readonly IBlobClient _blobClient;
        private readonly IConsumptionService _consumptionService;

        public EventHubTrigger(
            IConsumptionService consumptionService,
            IBlobClient blobClient)
        {
            _consumptionService = consumptionService;
            _blobClient = blobClient;
        }

        [Function(nameof(EventHubTrigger))]
        public async Task HandleEventHubEvents(
            [EventHubTrigger(
                "hub-consumptions",
                ConsumerGroup = "$Default",
                Connection = "EventHubConnection",
                IsBatched = true)]
            byte[][] events,
            FunctionContext executionContext)
        {
            var payloadSplit = events
                .Select(e => ConsumptionMessage.Open(e))
                .ToList();

            await SavePayloadsAsync(payloadSplit);
        }

        private async Task SavePayloadsAsync(
            IReadOnlyList<ConsumptionMessage> payloads)
        {
            foreach (var payload in payloads)  {
                var blobBytes = await _blobClient
                    .DownloadBlobAsync(new Uri(payload.WriteLocation));
                var blobPayload = ConsumptionMessage
                    .Open(blobBytes);
                await _consumptionService
                    .CreateAndSaveConsumptionAsync(blobPayload);
            }
        }
    }
}
```

Listing 28. Processing Event hub events in Azure Function.

Event hub trigger is almost the same as Service Bus trigger. Only the [EventHubTrigger] parameters are different, but they essentially function the same.

Last trigger would have been an Event Grid trigger, but at this point in time it did require an additional setup on the schema side to get it work. It still utilized JSON on the schema level, while the data property value could be the only Protobuffed part. It was also impossible to get it easily debugged, whereas the two other types of triggers worked flawlessly. Addition to this, Event Grid trigger must be defined for a resource that is already deployed, so a trigger could not have been defined until the function were deployed. All these things considered, the Event Grid is deemed to be too challenging, and therefor not adequate for the usage as a final pick. This will conclude Event Grid implementations and nothing further is done with it.

Only thing left, is to create the IConsumptionService that will handle timeseries transformations to the database model and sending a new type of consumption saved message, that can be picked up by EnerKey post processes.

Due to most of things in this implementation are considered as a trade secret for EnerKey, only interface-level was allowed to be shared.

```
namespace EnerKey.Consumption.Functions
{
    using System.Threading.Tasks;
    using Api.Messaging.Standard.Consumption.V1;
    public interface IConsumptionService
    {
        Task HandleConsumptionsAsync(ConsumptionMessage message);
    }
}
```

Listing 29. IConsumptionService interface.

This IConsumptionService is responsible of creating a standardized consumption based timeseries model that will be saved to InfluxDB. These were explained in the chapter 2.1 and 2.2. As timeseries must be saved as an interval version, a cumulative data within the message requires an additional step. It was not allowed to disclose any further information about it here.

After IConsumptionService has saved the timeseries to InfluxDB, a new message type is used to send a separate message forward to next channel. This message will then be picked up by existing post processes EnerKey has.

4.4 EnerKey.Api.Influxion

EnerKey.Api.Influxion is an abstraction layer on top of the InfluxData official C# client library. As EnerKey plans to migrate existing solutions to the 2.0 or newer version of InfluxDB, they did not have anything ready yet. EnerKey's old influx client only works for older InfluxDB versions and it is built on top of another third-party library. After analysing the official InfluxData client library for new versions of InfluxDB, it seemed a bit too broad for EnerKey. EnerKey likes to use tailored versions, that look and feel like any other major libraries they use or have built on top of. Therefor a new abstraction layer on top of the official client, is required.

This library should be publicly available in the GitHub. Creating the library with open source, EnerKey can attract other people around to world to also contribute and improve it. There is certain attraction to it, as its usage is different from the official client and reminds of more traditional ORM, such as Entityframework Core.

This EnerKey.Api.Influxion library was implemented as a .NET5.0 library and uses InfluxDB.Client as its base. Library should remind and feel like Entityframework Core. As the final library size grow to a quite huge, only the most important details with it are highlighted. As the time also run out, some parts of the library were unfinished.

4.4.1 InfluxClient

At its core, InfluxClient is a sole wrapper for the InfluxData's InfluxDBClient, that provides all the functionality against InfluxDB.

```
namespace EnerKey.Api.Influxion
{
    ...

    public interface IInfluxClient
    {
        public Task WriteAsync<TMeasurement>(
            IReadOnlyCollection<TMeasurement> entities,
            CancellationToken cancellationToken = default)
            where TMeasurement : class;

        public Task<List<TMeasurement>> QueryFluxAsync<TMeasurement>(
            string fluxQuery,
            CancellationToken cancellationToken = default)
            where TMeasurement : class;

        public Task<bool> PingAsync(
            CancellationToken cancellationToken = default);

        public Task<Bucket> CreateBucketAsync(
            string name,
            BucketRetentionRules retention,
            CancellationToken cancellationToken = default);
    }
    ...
}
```

Listing 30. IInfluxClient wrapper.

As can seen from the listing 30, IInfluxClient is the interface that defines all the methods EnerKey can use to interact with the InfluxDB. At the time of writing, there is capability of writing data asynchronously to the InfluxDB. Flux-query language can be used to read data from the InfluxDB. Lastly, one can test if InfluxDB is up and create a new bucket, that will contain the actual data.

4.4.2 InfluxDbContext

Next up, abstract class called InfluxDbContext is created.

```
namespace EnerKey.Api.Influxion
{
    using System.Threading;
    using System.Threading.Tasks;
    using InfluxDB.Client.Api.Domain;

    public abstract class InfluxDbContext
    {
        public readonly IInfluxClient Client;

        protected InfluxDbContext(IInfluxClient client)
        {
            Client = client;
        }

        protected InfluxMeasurement<TMeasurement> Set<TMeasurement>()
            where TMeasurement : class, new()
        {
            return new InfluxMeasurement<TMeasurement>(this);
        }

        public Task<bool> PingAsync()
        {
            return Client.PingAsync();
        }

        public async Task<Bucket> CreateBucketAsync(
            string name,
            BucketRetentionRules retention = null,
            CancellationToken cancellationToken = default)
        {
            return await Client.CreateBucketAsync(
                name,
                retention ?? new BucketRetentionRules(
                    BucketRetentionRules.TypeEnum.Expire,
                    3600,
                    3600),
                cancellationToken);
        }
    }
}
```

Listing 31. InfluxDbContext class.

This class serves as a base for user created contexts and defines a set of generic TMeasurements as InfluxMeasurement. TMeasurement is a class, containing all the keys and fields of the data stored in the InfluxDB. InfluxDbContext also contains a database level method, that can be used directly, for example: ping the database.

4.4.3 InfluxMeasurement

InfluxMeasurement contains all the methods that a data wishes to achieve with InfluxDB. It takes a generic TMeasurement and functions the same for all of those.

```
namespace EnerKey.Api.Influxion
{
    using System.Collections.Generic;
    using System.Threading;
    using System.Threading.Tasks;

    public interface IInfluxMeasurement<TMeasurement>
        where TMeasurement : class, new()
    {
        public Task WriteAsync(
            IReadOnlyCollection<TMeasurement> entities,
            CancellationToken cancellationToken = default);

        public Task<List<TMeasurement>> QueryFluxAsync(
            string fluxQuery,
            CancellationToken cancellationToken = default);
    }

    public class InfluxMeasurement<TMeasurement>
        : IInfluxMeasurement<TMeasurement>
        where TMeasurement : class, new()
    {
        private readonly InfluxDbContext _influxDbContext;
        public InfluxMeasurement(InfluxDbContext context)
        {
            _influxDbContext = context;
        }

        public Task<List<TMeasurement>> QueryFluxAsync(string fluxQuery,
            CancellationToken cancellationToken = default)
        {
            return _influxDbContext.Client
                .QueryFluxAsync<TMeasurement>(fluxQuery, cancellationToken);
        }

        public Task WriteAsync(
            IReadOnlyCollection<TMeasurement> entities,
            CancellationToken cancellationToken = default)
        {
            return _influxDbContext.Client
                .WriteAsync(entities, cancellationToken);
        }
    }
}
```

Listing 32. IInfluxMeasurement and its implementation.

As can be seen from the listing 32, at the time of writing, a possibility of writing the data was implemented. Data can be read using flux-query language.

4.4.4 InfluxBucketAttribute

Couple attributes to ease the usage of the measurements, are needed.

```
namespace EnerKey.Api.Influxion.Attributes
{
    using System;

    /// <summary>
    ///     Set bucket where you want to write the measurement.
    ///     Use <see cref="IsTenanted"/> if you wish to use tenanting with the
measurement
    /// </summary>
    [AttributeUsage(AttributeTargets.Class)]
    public class InfluxBucketAttribute : Attribute
    {
        public string BucketName { get; }
        public bool IsTenanted { get; }

        public InfluxBucketAttribute(string bucketName)
        {
            BucketName = bucketName;
            IsTenanted = false;
        }

        public InfluxBucketAttribute(string bucketName, bool isTenanted)
        {
            BucketName = bucketName;
            IsTenanted = isTenanted;
        }
    }
}
```

Listing 33. InfluxBucketAttribute implementation.

This InfluxBucketAtrribute allows setting up the bucket where the measurement is saved and if it is an entity that can be tenanted. Tenanting in EnerKey means that can it be isolated to a specific customer, while hiding it completely from the rest of the customers.

4.4.5 InfluxWritePrecisionAttribute

As InfluxDB supports optimizations for writing the data, whenever a certain time precision is required, the precision should be crudest possible.

```
namespace EnerKey.Api.Influxion.Attributes
{
    using System;
    using InfluxDB.Client.Api.Domain;

    /// <summary>
    ///     Use to set <see cref="InfluxDB.Client.Api.Domain.WritePrecision"/>
for the class.
    ///     If attribute is not used, default precision will be used on writes
    /// </summary>
    [AttributeUsage(AttributeTargets.Class)]
    public class InfluxWritePrecisionAttribute : Attribute
    {
        public WritePrecision WritePrecision { get; }

        public InfluxWritePrecisionAttribute(WritePrecision precision)
        {
            WritePrecision = precision;
        }
    }
}
```

Listing 34. InfluxWritePrecisionAttribute and its implementation.

With this attribute, WritePrecision can be defined in a measurement level. There can then be different types of data with various precisions. On the other hand, with this attribute the precision must be set once and then it can be forgotten as it is handled within the Influxion library thereafter.

4.4.6 Library usage

This subchapter introduces the end result on how the library is used to write certain data to InfluxDB.

```
namespace EnerKey.Api.Influxion.Tests.Entities
{
    ...
    [InfluxBucket("influxion_tests")]
    [InfluxWritePrecision(WritePrecision.S)]
    [Measurement("timelesstestentity")]
    public class TestEntity
    {
        [Column("testid", IsTag = true)]
        public string TestId { get; set; }

        [Column(IsTimestamp = true)]
        public DateTime Timestamp { get; set; }

        [Column("value")]
        public double Value { get; set; }
    }
}
```

Listing 35. TestEntity class.

Here a measurement, called TestEntity, is defined. It contains an TestId that acts as a tag for InfluxDB measurement. Then there is Timestamp that represents when the value for the tag occurred. Lastly, there is Value that acts as a column containing a numeric value that occurred at the timestamp concerning the tag as can be seen from listing 35.

```
namespace EnerKey.Api.Influxion.Tests
{
    using Entities;
    public class TestContext : InfluxDbContext
    {
        public InfluxMeasurement<TestEntity> Test => Set<TestEntity>();

        public TestContext(IInfluxClient client) : base(client)
        {
        }
    }
}
```

Listing 36. TestContext that inherits InfluxDbContext.

Next, TestContext inherits the InfluxDBContext. This TestContext defines a single InfluxMeasurement with TestEntity as its generic type.

Now the test setup is complete, and an actual test can be written.

```
namespace EnerKey.Api.Influxion.Tests
{
      ...
    [Collection(SingleHostCollection.CollectionName)]
    public class ContextTests
    {
        private readonly InfluxionFixture _fixture;
        public ContextTests(InfluxionFixture fixture)
        {
            _fixture = fixture;
        }

        [Fact]
        public async Task Should_Write_To_Measurement()
        {
            var context = _fixture.GetService<TestContext>();
            await context
                     .CreateBucketAsync(nameof(Should_Write_To_Measurement));

            var ts = DateTime.UtcNow;
            await context.Test.WriteAsync(
                new List<TestEntity>
                {
                    new()
                    {
                        TestId = "Abc",
                        Value = 1.0,
                        Timestamp = ts,
                    },
                });

            var savedData = await context
                     .Test.QueryFluxAsync("...");
            Assert.NotNull(savedData);
            Assert.NotEmpty(savedData);
            Assert.True(savedData.Exists(d =>
                    d.TestId == "Abc"
                    && d.Value == 1.0
                    && d.Timestamp == ts));
        }
    }
}
```

Listing 37. XUnit test to check that data is written to InfluxDB.

Here an integration test is defined. Then it gets the TestContext from the Fixture, and creates a bucket and writes a dummy data to InfluxDB. Data is then fetched from the database and asserted. Assertion is that data was what was saved before.

These were the key points with the Influxion. There were still a lot of things to do for this library. Hopefully it would be publicly available from the GitHub at the end of first quarter in the 2022.

5 Conclusions and discussions

EnerKey had a list of requirements:

- Cloud native approach to handle energy consumption data
- Allow scaling
- Fix design flaws of the old system
- Be secure, highly available and fault tolerant
- Be performant
- Up to EnerKey quality standards for written software
- Gather information about Azure messaging solutions
- A product that can be used to replace the old solution

Thesis resulted in a product, that utilizes Azure Messaging with Azure Storage and InfluxDB. Final product contains C# code implementations that can be used as a basis for replacing the EnerKey's old system. Utilizing Azure messaging, EnerKey can achieve cloud native approach. Introduced Azure products are highly available and secure.

By creating a new messaging library and Influxion library, EnerKey can replace the old system with less complexity. Code was written in a way that was up to EnerKey today's standards. Utilizing protobuf, EnerKey can achieve more performance.

Allow scaling is still to be determined. Time run out and another EnerKey developer jumped on board to finalize the product and build more on top of it. This resulted in a stop of implementation and therefor it is unsure how the solution scales. First thing for EnerKey to do next, is to ensure the scalability by utilizing load testing and different data scenarios.

Fault tolerance was left out. Most likely EnerKey will later need more implementations to allow fault tolerance. While using Azure services (e.g., Azure Service bus) they have high SLA promises from Microsoft. At least in basic cases that is enough to satisfy EnerKey's own requirements for service incidents.

With verbal communication, knowledge about Event Grid and Event Hub has been shared with other EnerKey employees. EnerKey now has basic knowledge and quirks these services have.

Applying a modern messaging theorem and combining it with Azure Products can be applied to many other types of timeseries, not just the energy consumptions. Thesis solution is applicable for other systems as well that can utilize Azure and must transmit a lot of data at once.

Overall, the results and achievements are satisfying. Subject was interesting. This was a long journey that started years ago. When the old implementation was made, EnerKey assigned couple parts of it for me to write. As the time passed, it became apparent to EnerKey that old system is not adequate. Same time it had become one complex application that was solving problems with incorrect approaches due to the fact that it was not explicitly created for those. This thesis was a perfect fit to fix all those issues. On the start, it seemed that only messaging need to be solved to create a better system. As the work progressed, it only then become apparent that the old Influx library EnerKey has, is not going to be usable for this implementation. This was a surprising discovery. It negatively impacted the work itself as an increase of about third of the work was now required. It led to substantially more things to be done, than was originally planned.

Because there was a lot of things to do in the end, a cutting of the details had to be made and only write about the most important key points, then drop the rest from the thesis document. This ended up reducing the level of details written in the thesis. It was not satisfying thing to do. The devil is in the details, they say. Here all those super detailed parts can be found on the final product handed over to EnerKey, but not on this written document. Writing more details would have led to more approachable document, but the size would have been completely different.

Now on the hindsight, InfluxDB specific details should have been dropped completely out and concentrate full on about the messaging alone. In a sense,

this would have not been possible as there was a clear agreement with specific goals set by EnerKey for this work. Nevertheless, goals were set and goals were met. Thesis still clearly benefitted the EnerKey. In a product sense, this was perfect outcome.

Thesis has other benefits too. Here is a modern take on enterprise level messaging and how that applies in practice. A lot of knowledge was gain. Also new types of Azure products were introduced. Overall, this has been a great learning success and will surely be beneficial in the years to come.

For you, the reader, hopefully this thesis was able to give you a new thoughts and ideas, and introduce a new concepts that will allow you to achieve something great.

References

1 E. Clower. Introduction to the Fundamentals of Time Series Data and Analysis [Internet]. 13 Sep 2019 [updated 12 Oct 2021; cited 29 Oct 2021]. Available from: https://www.aptech.com/blog/introduction-to-the-fundamentals-of-time-series-data-and-analysis

2 InfluxData. What is time series data [Internet]. 2021 [cited 6 Sep 2021]. Available from: https://www.influxdata.com/what-is-time-series-data

3 Sunpower Electronics. Peak Power [Internet]. 2019 [cited 6 Sep 2021]. Available from: https://www.sunpower-uk.com/glossary/what-is-peak-power

4 G. Hohpe, B. Woolf. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions [E-book]. 2003 [cited 10 Sep 2021]. Chapter 2. Available from: https://learning.oreilly.com/library/view/enterprise-integration-patterns/0321200683/ch02.html

5 Microsoft. Messaging services on Azure [Internet]. 2021 [cited 13 Sep 2021]. Available from: https://azure.microsoft.com/en-us/solutions/messaging-services/#products

6 Microsoft. What is Azure Relay [Internet]. 2 Feb 2021 [updated 6 Sep 2021; cited 13 Sep 2021]. Available from: https://docs.microsoft.com/en-us/azure/azure-relay/relay-what-is-it

7 Microsoft. Azure SignalR Service [Internet]. 2021 [cited 13 Sep 2021]. Available from: https://azure.microsoft.com/en-us/services/signalr-service/#overview

8 Microsoft. Azure HDInsight [Internet]. 2021 [cited 13 Sep 2021]. Available from: https://azure.microsoft.com/en-us/services/hdinsight/#overview

9 Microsoft. Notification Hubs [Internet]. 2021 [cited 13 Sep 2021]. Available from: https://azure.microsoft.com/en-us/services/notification-hubs/#overview

10 Microsoft. Azure IoT Hub [Internet]. 2021 [cited 13 Sep 2021]. Available from: https://azure.microsoft.com/en-us/services/iot-hub/#overview

11 Microsoft. Azure IoT Hub pricing [Internet]. 2021 [cited 13 Sep 2021]. Available from: https://azure.microsoft.com/en-us/pricing/details/iot-hub/

12 Microsoft. Azure Web PubSub [Internet]. 2021 [cited 13 Sep 2021]. Available from: https://azure.microsoft.com/en-us/services/web-pubsub/#overview

13 Microsoft. Event Grid [Internet]. 2021 [cited 13 Sep 2021]. Available from: https://azure.microsoft.com/en-us/services/event-grid/#overview

14 Microsoft. Event Hub [Internet]. 2021 [cited 13 Sep 2021]. Available from: https://azure.microsoft.com/en-us/services/event-hubs/#overview

15 Microsoft. Service Bus queues, topics, and subscriptions [Internet]. 27 Aug 2021 [updated 28 Aug 2021; cited 13 Sep 2021]. Available from: https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-queues-topics-subscriptions

16 Microsoft. Get Started with Azure Queue Storage using .NET [Internet]. 8 Sep 2020 [updated 18 May 2021; cited 14 Sep 2021]. Available from: https://docs.microsoft.com/en-us/azure/storage/queues/storage-dotnet-how-to-use-queues?tabs=dotnet

17 K. Grochowski, M. Breiter, R. Nowak. Serialization in Object-Oriented Programming Languages [Internet]. 8 Aug 2019. [cited 17 Sep 2021]. Available from: https://www.intechopen.com/chapters/68840

18 B. Smith. Beginning Json [E-book]. Mar 2015 [cited 17 Sep 2021]. Chapter 4. Available from: https://learning.oreilly.com/library/view/beginning-json/9781484202029/9781484202036_Ch04.xhtml

19 Google. Protobuf [Internet]. [Year unknown] [cited 17 Sep 2021]. Available from: https://opensource.google/projects/protobuf

20 Microsoft. Service Bus pricing [Internet]. 2021 [cited 18 Sep 2021]. Available from: https://azure.microsoft.com/en-au/pricing/details/service-bus

21 Microsoft. Azure Event Hubs quotas and limits [Internet]. 11 May 2021 [updated 20 Jun 2021; cited 18 Sep 2021]. Available from: https://docs.microsoft.com/en-us/azure/event-hubs/event-hubs-quotas

22 Microsoft. Azure Event Grid quotas and limits [Internet]. 17 Feb 2021 [cited 18 Sep 2021]. Available from: https://docs.microsoft.com/en-us/azure/event-grid/quotas-limits

23 Microsoft. Azure Blob Storage [Internet]. 2021 [updated 23 Sep 2021; cited 27 Sep 2021]. Available from: https://azure.microsoft.com/en-us/services/storage/blobs/#features

24 Microsoft. Manage blob properties and metadata with .NET [Internet]. 25 Sep 2020 [updated 23 Sep 2021; cited 27 Sep 2021]. Available from: https://docs.microsoft.com/en-us/azure/storage/blobs/storage-blob-properties-metadata?tabs=dotnet

25 Microsoft. Hot, cool, and archive access tiers for blob data [Internet]. 24 Sep 2021 [updated 28 Sep 2021; cited 29 Sep 2021]. Available from: https://docs.microsoft.com/en-us/azure/storage/blobs/access-tiers-overview

26 Microsoft. Optimize costs by automatically managing the data lifecycle [Internet]. 18 Aug 2021 [updated 29 Sep 2021; cited 30 Sep 2021]. Available from: https://docs.microsoft.com/en-us/azure/storage/blobs/lifecycle-management-overview?tabs=azure-portal

27 InfluxData. Time series database (TSDB) explained [Internet]. 2021 [cited 6 Oct 2021]. Available from: https://www.influxdata.com/time-series-database

28 InfluxData. InfluxDB [Internet]. 2021 [cited 6 Oct 2021]. Available from: https://www.influxdata.com/products/influxdb

29 Microsoft. Azure Functions [Internet]. 2021 [cited 14 Oct 2021]. Available from: https://azure.microsoft.com/en-us/services/functions

30 Protobuf-net [Internet]. 2021 [cited 14 Oct 2021]. Available from: https://github.com/protobuf-net/protobuf-net