# Team Azure Final report

Table of Contents

[Team Azure Final report 1](#_Toc122277288)

[Team 3](#_Toc122277289)

[Project goals 4](#_Toc122277290)

[Design Principles 4](#_Toc122277291)

[Architecture 4](#_Toc122277292)

[5](#_Toc122277293)

[Frontend layer 6](#_Toc122277294)

[Messaging layer 7](#_Toc122277295)

[Distributed layer 10](#_Toc122277296)

[Data layer 12](#_Toc122277297)

[Logging 13](#_Toc122277298)

[Infrastructure-as-code 14](#_Toc122277299)

[Functionalities 15](#_Toc122277300)

[Shared distributed state 15](#_Toc122277301)

[Naming and node discovery 15](#_Toc122277302)

[Fault Tolerance 16](#_Toc122277303)

[Synchronization and consistency 17](#_Toc122277304)

[18](#_Toc122277305)

[Scalability and performance 18](#_Toc122277306)

[The key enablers and the lessons learned during the development of the project. 20](#_Toc122277307)

[Group member participation 20](#_Toc122277308)

## Team

* Jari Sokka
* Jukka Koskelin
* Ville Muilu

## Project goals

The purpose of the project was to learn to understand the details of a distributed system by implementing a small-scale/simple webshop with public cloud Platform-as-a-Service (PaaS) components. We chose Microsoft Azure as the platform as one of project member had previous experience on using it, though not actually implementing a distributed system. At the same time, this gave the other members of the group a good opportunity to familiarize themselves with Azure and the functionalities it offers.

Although Azure offers functionality that even implements the features of the distributed system automatically for the user, it does not automatically make the implementation of the project easier. Using Azure meant additional work in terms of configuring system settings and coordinating functionalities. This was known to bring a challenge to the implementation of the project, but since one of the group members was familiar with the system, we believed we would overcome this challenge.

## Design Principles

We set on exploring what PaaS services would fit the project, with clear understanding the architecture we ended up with might not be one we designed at the start. We did, however, have quite a clear though of the architectural layers we were going to implement and the languages to be used. The layers are:

* Frontend layer, implemented as a single page web app.
* Messaging layer, implemented with Azure Service Bus and Event Grid. This layer has no programming implementation (unless we count IAC).
* Distributed layer, implemented as Azure Functions. This is the main distributed layer of the software and consists of identical Azure Function Apps deployed to different Azure Regions.
* Data layer, implemented with Azure Cosmos DB and Azure Function Apps hosting a simple dbApi. Originally this layer was designed as to be deployed in single region with only one dbApi, but we ended up deploying all services to same regions as the distributed layer, to be able to test the consistency features of Cosmos DB.

### Architecture

In whole, the architecture pattern used can be best though as a modular monolith - a service made of multiple components that can be changed to use different services if needed, but not implementing a microservice pattern where, for example, each node would be represented by a self-sustained service with would handle the operations for its own domain. While we do deploy a complete set of services to several regions under the messaging layer, each set of these services consist of similar services with just region-specific configurations, so they are more of a (arguably somewhat incomplete) fault tolerance features than microservices as such.

### Chart Description automatically generated

Figure . Overall architecture

Note that Azure Storage accounts are not drawn in the architecture diagram. Every Azure Function app is created with a paired storage account, where the function apps store their internals, like the deployed source code. We simply use the same storage account for storing data in a separate table service.

### Frontend layer

The frontend layer representing a simple webshop was to be implemented as a single page app (SPA) with React framework that would fetch data for the webshop from the data layer, and initiate updates by send asynchronous messages to the middleware messaging layer.

We chose Azure Static Web Apps (SWA) as the PaaS service to be used and had no reason to change it during the implementation phase, even if we did stumble into several limitations concerning the integration with messaging layer. SWA is actually made of two different services; hosting of the single page application and hosting of a backend function, which is a limited version of an Azure Function.

One of the limitations is that the backend function only supports http-binding for the function, and it is mainly meant to be used between the SPA and the backend function itself. Implementing a messaging integration, however, requires an outbound binding to the messaging layer, which the backend function does not support (which is not really documented well).

We went around this limitation by implementing the messaging integration by using Azure nodejs npm-package for the messaging server, which is a bit of a messy way of implementing the integration but works well enough for demonstration purposes. The more elegant way would have been not to deploy the backend function at all, but either to implement the integration directly to the SPA, or by deploying a separate Azure Function App to handle the messaging integration (which would have required a premium offering of the SWA).

Integration to service bus is crated from the backend api with service bus sdk:

const { ServiceBusClient } = require("@azure/service-bus");  
  
...  
  
 const sbClient = new ServiceBusClient(process.env.SBconnectionString);  
 const sender = sbClient.createSender(process.env.queueName);  
   
 const message = { ean: req.body.task.ean, name: req.body.task.name, amount: req.body.task.amount }

Message from frontend to service bus in formatted like this:

{ ean: 2222, name: 'gift two', amount: 2 }

The frontend retrieves product amount directly from the database or, more precisely, from its API. The request is a standard http call.

export async function getAllProducts() {  
 const response = await fetch('https://func-distributed-dbapi-we- 001.azurewebsites.net/api/db', {  
 method: 'GET',  
 headers: {'Content-Type': 'application/json --verbose'},  
 });  
 return await response.json();  
}

References:

* React single page app can be found under /front -folder in repo: <https://github.com/JukkaK/Distributed2022/tree/main/front>.
* SPA backend function can be found under /api -folder in repo: <https://github.com/JukkaK/Distributed2022/tree/feature/main/api>.
* No IAC - implementation for SPA, since it’s not in the scope of project as such (and implementing static web apps with Bicep has some issues).
* Static Web App docs: <https://learn.microsoft.com/en-us/azure/static-web-apps/>
* Github issue with explanations and partial solution for function bindings: <https://github.com/Azure/static-web-apps/issues/141>

### Messaging layer

Selecting the PaaS service for the messaging layer was, from architectural viewpoint, one of the most demanding tasks of the project. Azure basically has two services that can both handle the messaging: *Azure Event Grid* which is meant for doing integration with lightweight messages, and *Azure Service Bus* which is meant for business-critical messages with possibility of larger payloads.

We ended up using *Service Bus* as it implements a lot of the features required from the project. With Service Bus, we use a simple *queue* where frontend application pushes messages, and to which our distributed worker nodes subscribe to. Our Service Bus queue uses a first in, first out (FIFO) paradigm, which means that it’s pretty dumb; messages pushed to the queue retain their order of appearance, and they are served out in that same order. Subscribed services consume the messages with the fastest-wins -principle.

Service Bus supports even more fault tolerant messaging options, like peek-locking, meaning that a subscribed service first reads a message from the queue and reserves it so that competing services are not able to read it. After completing its transaction, it then deletes the message from the queue, and if the transaction for some reason fails, queue releases the message back for others to consume after a configurable period of time. Unfortunately our chosen implementation language (javascript) did not support this feature yet.

The relevant part of service bus message format:

bindingData: {  
 invocationId: '0175dcfa-9e7f-4269-9889-0777e17b2e89',  
 messageReceiver: {},  
 messageSession: {},  
 messageActions: {},  
 sessionActions: {},  
 receiveActions: {},  
 client: {  
 fullyQualifiedNamespace: 'sb-distributed-we-001.servicebus.windows.net',  
 isClosed: false,  
 transportType: 0,  
 identifier: 'sb-distributed-we-001.servicebus.windows.net-b3b7489b-e52a-4afc-a66c-2b009b03422b'  
 },  
 deliveryCount: 1,  
 lockToken: '44863086-e54f-45a8-b8fa-993d46434c80',  
 expiresAtUtc: '2022-12-14T22:36:50.52',  
 expiresAt: '2022-12-14T22:36:50.52+00:00',  
 enqueuedTimeUtc: '2022-12-14T20:36:45.52',  
 enqueuedTime: '2022-12-14T20:36:45.52+00:00',  
 messageId: 'e3500ddd88cf41d385c6f8944f8a3554',  
 sequenceNumber: 310,  
 applicationProperties: {},  
 userProperties: {},  
 ean: 2222,  
 name: 'gift two',  
 amount: 1  
 }

Where *userProperties*-block contains the actual message payload. We also extract *messageId*, which is generated by Service Bus, and use it in the serviceWorker -nodes shared state item.

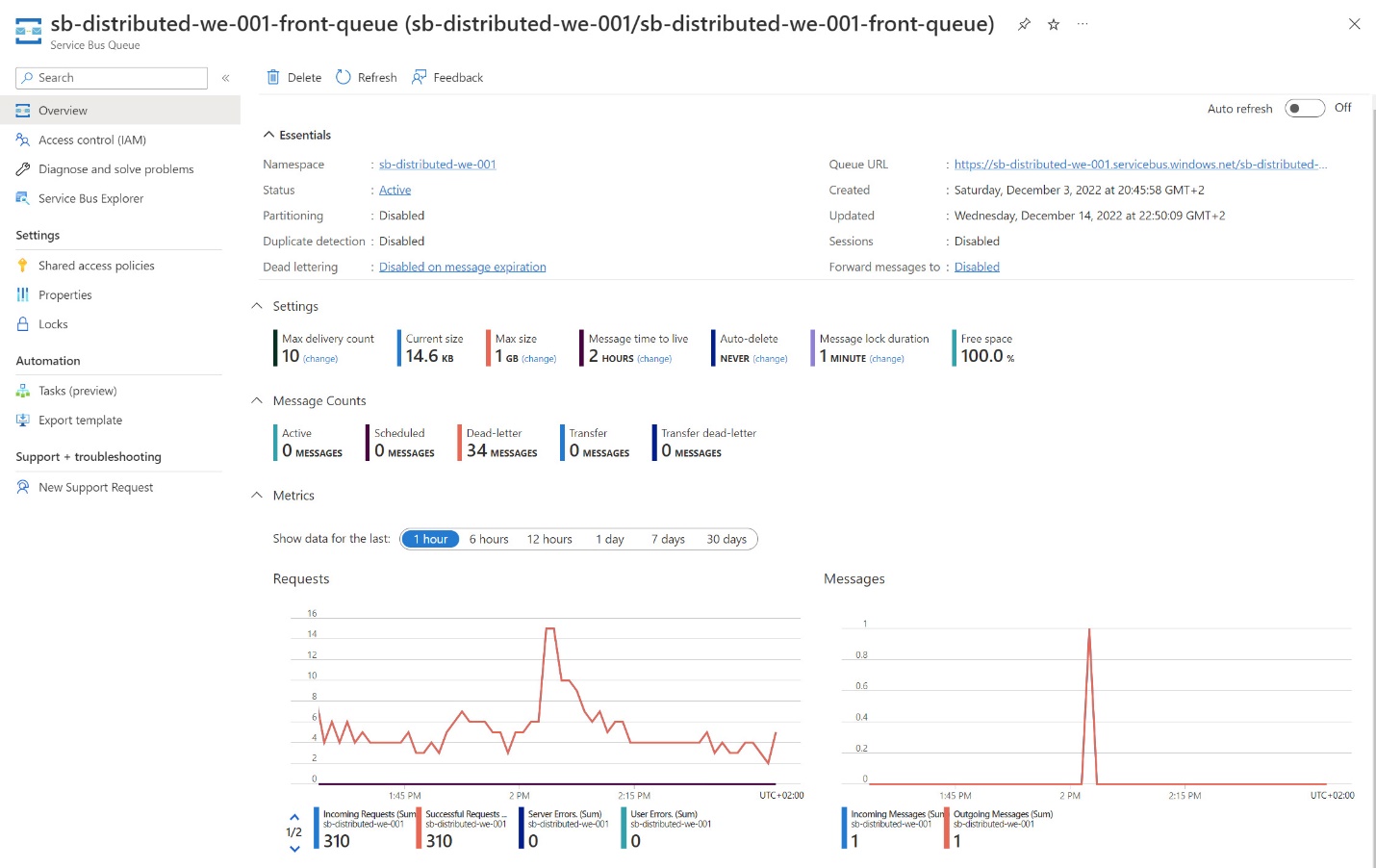


Figure . Azure offers a comprehensive selection for viewing and managing a Service Bus messages.

In the end we implemented an Event Grid Topic as well, for node-to-node messaging. When a serviceWorker node has successfully called dbApi for an update operation, it composes an event with the messageId of the original message as payload and sends it to Event Grid. Every serviceWorker node is subscribed to Event Grid Topic, so every new event triggers a function that stores the received messageId to a table in storage account tied to the function. The serviceWorker which originally handled the message stores the messageId as well, so all the functions have the same data stored in their tables (as long as every node is running - a stopped function app does store anything).

Graphical user interface, application, email

Description automatically generated

Figure . Subscriptions are created in Azure Event Grid Topic

* Service Bus has no application implementation, only IAC-implementation.
* Azure messaging service comparison: <https://learn.microsoft.com/en-us/azure/event-grid/compare-messaging-services>
* Service bus queues and topics: <https://learn.microsoft.com/en-us/azure/service-bus-messaging/service-bus-queues-topics-subscriptions>
* Service bus receive operations: <https://learn.microsoft.com/en-us/azure/service-bus-messaging/message-transfers-locks-settlement>

### Distributed layer

Diagram

Description automatically generated

Figure . Message flow

We chose our transaction layer to be the distributed part of the system and ended up having three of distributed worker nodes that consume messages from messaging layer and perform the update operations against the data layer. We have a single worker implementation and deploy it to multiple Azure Function Apps that are geographically distributed. Worker application has an inbound Service Bus binding that is configured to listen to the Service Bus Queue. When messages appear in the queue, a function is triggered, the function reads the message and subsequently calls the data layer to perform an update transaction with the details gathered from the message payload. After completion the function stops, and is triggered again when the next message is picked up from the queue.

Function that triggers when message comes to the Service Bus Queue looks like this:

import { AzureFunction, Context } from "@azure/functions"  
...  
const serviceBusQueueTrigger: AzureFunction = async function(context: Context, mySbMsg: any): Promise<Object> {  
 await axios({  
 method: 'PUT',  
 url:process.env["DBAPI\_URL"],   
 data: {mySbMsg}  
...

Worker receives data from service bus in mySbMsg object, where the payload simply is:

{ ean: 2222, name: 'gift two', amount: 2 }

After submitting the call to dbApi, function composes an event that it sends out via the functions outbound binding to Event Grid Topic.

            context.bindings.outputEvent = {

                id: 'message-id',

                subject: 'subject-name',

                dataVersion: '1.0',

                eventType: 'event-type',

                data: context.bindingData.messageId,

            };

Every serviceWorker has a second function that is triggered by new events in Event Grid Topic. This function simply stores the messageId received from the event into a table in associated storage account.

const tableClient = TableClient.fromConnectionString(process.env.AzureWebJobsStorage, "state");

    await tableClient.createTable();

    const entity = {

        partitionKey: "p1",

        rowKey: eventGridEvent.data,

        date: new Date()

      };

      await tableClient.createEntity(entity);

Graphical user interface, text, application

Description automatically generated

Figure . Stored messageIds in table storage

* Worker node implementation can be found under /serviceWorker-folder: <https://github.com/JukkaK/Distributed2022/tree/main/serviceWorker>
* Azure Functions docs: <https://learn.microsoft.com/en-us/azure/azure-functions/>
* Azure Functions service bus trigger implementation: <https://learn.microsoft.com/en-us/azure/azure-functions/functions-bindings-service-bus-trigger?tabs=in-process%2Cextensionv5&pivots=programming-language-javascript>

### Data layer

We chose Azure Cosmos DB as our data storage option, as it is a global distributed service. We ended up deploying Cosmos DB in three regions in hopes of getting interesting results with consistency levels. That did not really happen as data remained woefully consistent despite our best efforts, like setting the consistency level to *‘eventual consistency’*.

Our Cosmos DB acts as a document storage with MongoDB API as implementation option. MongoDB was chosen because one of our team members was familiar with it. On top of it, we implemented a simple database api that also runs in an Azure Function application. As we decided to deploy the DB three regions, it made sense to deploy the apis to three regions as well, to ensure that all regions get writes.

DbApi receives simple API calls where the update request is in the body:

{  
 method: 'PUT',  
 url: 'https://func-distributed-dbapi-ne-001.azurewebsites.net/api/db',  
 originalUrl: 'https://func-distributed-dbapi-ne-001.azurewebsites.net/api/db',  
 headers: {  
  
...  
  
 body: { mySbMsg: { ean: 2222, name: 'gift two', amount: 2 } },  
 rawBody: '{"mySbMsg":{"ean":2222,"name":"gift two","amount":2}}'  
}

The document representation of the data in Mongo DB nosql database is:

{  
 "\_id" : ObjectId("63928e20534bf30140349975"),  
 "ean" : 2222,  
 "pic" : "gift2.png",  
 "name" : "gift two",  
 "amount" : 21,  
 "updatedAt" : {  
 "$date" : 1670768206190  
 }  
}

* Cosmos DB implementation is IAC-only.
* DbApi implementation can be found under folder dbApi: <https://github.com/JukkaK/Distributed2022/tree/main/dbApi>
* Cosmos DB consistency levels: <https://learn.microsoft.com/en-us/azure/cosmos-db/consistency-levels>

### Logging

We use Azure Application Insights as to log and debug our function apps. Every function app is connected to the same Application Insight -instance and provide logging that goes through the layers, which the exception of the message layer, where we use Service Bus metrics to observe the amount of messages passing through the queue.

Graphical user interface, text, application, email

Description automatically generated

Figure . Trace logs in Applications Insights

### Infrastructure-as-code

The IAC implementation is in **/Infrastucture** -folder. It does not cover the whole solution (the static web app has been created via portal, as there tend to be some challenges with the application deployment if the resource is created with IAC, and also some additional log forwarding to cosmos db and service bus has been done manually).

Infrastructure\  
|---ai.bicep  
|---cosmosdb.bicep  
|---eventgrid.bicep  
|---funcapp.bicep  
|---funcappsbd.bicep  
|---law.bicep  
|---main.bicep  
|---servicebus.bicep

While the infra implementation is mostly out of the scope for the project, it is included in the repository as it’s easier to illustrate the node discovery configuration in the code.

The IAC code has been deployed via AZ CLI command line interface, so after logging in and setting the deployment context (ie. Azure subscription), deployment has been run with command in the Infrastructure folder:

az deployment create --location westeurope --template-file main.bicep

## Functionalities

### Shared distributed state

The distributed nodes (serviceWorkers) each send an event when updating an item in database. Every node, including the one sending the event, has subscribed to an Event Grid Topics and receive the event. Every node then stores the messageid from the event payload to their respective table storage, which then forms a shared state as every node knows the completed message history of the whole system. Every node also checks from their own store that the messageid of a message they receive is not already stored in the state-table. If it is, they don’t update the database.

The functionality was added as a kind of afterthought as we realized that we are missing some of the required features; the original plan was to distribute update events from the front end as well. As Event Grid Topics have less features that deal with message delivery than Service Bus, this kind of logic would have had much more use. In our case, Service Bus guarantees that only one node receives a single message, and the shared state can maybe be thought more as an audit log -feature.

### Naming and node discovery

The distributed nodes of the application are implemented as Azure Functions, which get their own unique address upon creation, like <https://func-distributed-backend-ne-001.azurewebsites.net>, and singular functions hosted in the same app can be found under */api/functionname*, with a function secret key appended after the url. So discovery, should there be any direct function-to-function -calls, in the simplest form would be just hardcoding the full urls of other functions with their secret keys, and you are good to go.

However, when using PaaS services as messaging layer, the usual way of node discovery is that nodes themselves subscribe to the messaging services. This is commonly done by using some sort of connection string that includes the secret key of the messaging service. In case of Service Bus, the connection string format is:

Endpoint=sb://sb-distributed-we-001.servicebus.windows.net/;SharedAccessKeyName=RootManageSharedAccessKey;SharedAccessKey=secretkeygoeshere

This of course means that when adding a new node to the system, the node has to know the connection string of the messaging service (and on function level, the name of the queue that the function binds to). To automate this phase of discovery, we have implemented creation of the service bus and worker nodes with infrastructure-as-code -approach. The service bus creation outputs the connection string and the iac module that creates function apps takes the connection string as an input parameter and creates a function app configuration key-value -pair of it.

output serviceBusConnectionString string = listKeys(serviceBusEndpoint, sb.apiVersion).primaryConnectionString

module workersNE 'funcapp.bicep' = {  
 scope: backendNeRg  
 name: 'backendNe'  
 params: {  
 location: 'northeurope'  
 cosmoscs: cosmosdb.outputs.cs  
 serviceBusConnectionString: servicebus.outputs.serviceBusConnectionString  
 aiKey: ai.outputs.aiKey  
 appName: 'backend'   
 }  
}

In real-life scenario, one should never pass secret values as outputs in iac-code, as they tend to show up in deployment logs. The current preferred way of doing this configuration in Azure would be not to use secret keys at all, but to enable a *managed identity* in the function apps, that would be authorized to access the Service Bus. A managed identity is basically a service identity created for a service in the Azure Active Directory Tenant used. But using classic connection strings is probably more illustrative way of doing this for the purposes of this project.

The connectivity between worker nodes and dbApis is implemented as simple url configuration like:

https://func-distributed-dbapi-we-001.azurewebsites.net/api/db

The connectivity between dbApi and Cosmos DB uses connection strings with a preference hint for the write region. Every regional dbApi has a hint set to the respective Cosmos DB region:

mongodb://cosmos-distributed-we-001:secretkey@cosmos-distributed-we-001.mongo.cosmos.azure.com:10255/?ssl=true&replicaSet=globaldb&retrywrites=false&maxIdleTimeMS=120000&appName=@cosmos-distributed-we-001@East US

### Fault Tolerance

In order to understand fault tolerance in Azure public cloud, there are several key concepts that need to be understood first.

**Regions** are the basic units of fault tolerance; they are geographical units of deployment and under the hood, they are made up of at least three paired datacenters in different physical locations. For example, the upcoming *Finland* region in Azure will have three physically separate datacenters in around western Uusimaa.

**Availability Zones** are the logical units inside a region, each made up of one or more datacenters. Finally, inside data centers there are **fault domains** and **upgrade domains** that make up the physical level of fault tolerance.

For the purposes of this project, we can assume that our services would survive anything short of a regional outage with minimal downtime, as our deployment scope for the resources that are not duplicated (service bus, static web apps, data layer) is the **West Europe** region. Other regions used are **North Europe** and **East US**.

More info of the basic region/zone design can be found from here:

<https://learn.microsoft.com/en-us/azure/reliability/availability-zones-overview>

Here are some fault tolerance figures and details we found for the services we used in this project. These numbers are rather difficult to find and the basis or details are not revealed to the user.

Azure Static Web App, where our frontend runs, is promised 99.95% availability.

Azure Cosmos DB provide 99.99% availability regardless of the number of regions associated with their database. Azure Cosmos DB automatically mitigates replica outages by guaranteeing at least three replicas of your data in each Azure region for your account within a four replica quorum.

Azure Service Bus is only given the promise that it is highly reliable and it is said that typically, an outage does not cause loss of messages or other data. In Premium service there is an option to use Availability Zones and it also includes feature where Service Bus manages three copies of messaging store (1 primary and 2 secondary).

Azure Event Grid is promised 24-hour retry with exponential backoff to make sure events are delivered. They are also natively spread across multiple fault domains in every region.

In Function App there is no built-in redundancy available. However to avoid loss of execution during outages, user can redundantly deploy the same functions to function apps in multiple regions. When running the same function code in multiple regions, there are two patterns user can chose from: active/active or active/passive. Our functions are in active/active state so they are all actively running. In order for this to work in the best possible way, [one of the load balancing Azure services](https://learn.microsoft.com/en-us/azure/architecture/guide/technology-choices/load-balancing-overview) should be implemented. This would coordinate traffic between all areas/functions. We do not use this service because the service bus subscriptions work well enough; if a function is stopped, there are still the functions deployed to two other regions that keep ingesting the messages. However, for real fault tolerance, we should implement some kind of error handling for crashes that happen during an update operation. The best way to do this would be to use the peeklock-message settling in service bus.

### Synchronization and consistency

We implemented synchronization and consistency in the DB layer by deploying regional replicas to the database and playing with the consistency levels. This was supported by deploying both the worker nodes and dbApi to different regions, so we could be sure that every database replica would get writes. By setting the consistency level to ‘eventual consistency’ we actually hoped to see that the replicas would be out-of-sync from time to time, but we did not really witness this.

Other consistency levels are Strong, Bounded-staleness, Session and Consistent Prefix. So it would have been possible for us to choose Strong consistency, which would have been offered linearizability guarantee. This means serving requests concurrently.

Cosmos DB does not really have built-in tools for viewing and comparing the states of different replicas (or at least we did not find those) and given time we probably should have implemented regional frontends that would each have read and displayed the exact amounts of items in their respective database replicas. The only place where we could actually see something happening between different replicas was the view showing replication latency between regions.

## Graphical user interface, application, table, Excel Description automatically generated

Figure . Replication latency metrics

## Scalability and performance

Scaling in public cloud basically works in two different ways.

You can **scale up**, meaning that you add more computing power to the instance you currently have. Azure services usually have a *sku* (stock keeping unit) that one has to define when creating an instance of a service. Sku dictates multiple things; the amount of computing power (memory, etc) allocated to the service, the features available for the instance, and the price it costs - finding the correct sku is always a balancing act.

Scaling up and down is particularly hard for Azure Function Apps, as you are not only choosing the amount of computing power available. Function apps also have the concept of *plan*, which basically means how the function apps are hosted under the hood. The cheapest option is a *consumption* plan, where the functions are spun when they receive a call. Other options would be a *premium* or a *dedicated* plan, where the function apps would either have existing pre-warmed nodes all the time, or a dedicated web server with existing pre-warmed nodes. The distinction between consumption nodes and pre-warmed nodes is that there is a lag on the startup of the former. There is no way to automate scaling up; if you are using consumption plan and want to avoid the cold startup lag, you need to create new function app resources.

More about different plans and scaling can be found here:

<https://learn.microsoft.com/en-us/azure/azure-functions/functions-scale>

Other services we use also share the same idea of scaling up; they also get more features with skus that cost more, though with service bus and cosmos db scaling up also affects the throughput of the services.

You can also **scale out**, which means that your services can be run on multiple instances. Plenty of Azure PaaS services support some sort of autoscaling, meaning that some kind of metric-based thresholds can be configured into them, and if load goes over those thresholds, the services automatically create more instances where the load is run. For example, function apps, with both consumption and premium plan, scale up to 100 instances.

Another way of scaling out is to create multiple nodes of a service on your own, as we have done with the worker nodes; we deploy them to three separate Azure Regions (West and North Europe, and East US) which balances the load, but of course introduces more latency to the system as the messages have to travel across the globe. Each of these separate nodes could still scale out in their own, should we configure some kind of autoscaling for them.

With service bus, autoscaling would need a premium plan, but with that autoscaling can be configured in pretty much similar way.

See more about service bus namespace autoscaling can be found here:

<https://learn.microsoft.com/en-us/azure/service-bus-messaging/automate-update-messaging-units>

Static Web Apps don’t scale out as such, so they clearly are a bottle neck of our application. With Single Page Apps, it does not really make sense to scale up or out the resource hosting the compiled pages, but if there would be performance with the frontend, the correct way would be to implement some kind of caching and delivery -solution with it. Azure offers several services for that, including Azure CDN or Azure Front Door. Though even implementing those leaves us with the Static Web App backend functions, which really is an issue, as you can’t really do anything for those since they are a kind of under-the-hood-minimal-implementation of Azure Function App. Should there be any issues with performance of those backend functions, the correct way of addressing those would be to bump up the Static Web App sku to a paid standard-plan, which would allow us to host the backend functions in separate Azure Functions app that can be scaled as mentioned before.

For the Data layer, Azure Cosmos DB is by nature an autoscaling and globally distributed NoSql database. Cosmos DB uses a metric called *Request Units* (RU) to calculate all kinds of things, including the performance and pricing. As we initially had a single region deployment of both the Cosmos DB and the dBApi hosted in Azure Functions, the correct way of handling both the scaling and the availability of the services would be to deploy both in the same regions where the workers are deployed into, and configure the workers to use regional dbApis. This way, every region would work in a coherent way.

## The key enablers and the lessons learned during the development of the project.

The project gave a good idea of what distributed systems mean in practice. Although we did not implement the algorithms presented in the course during the project, we felt that we got more benefit by learning to use Azure’s PaaS-services.

We learned that services alone do not automatically provide a solution for building a distributed system. However, some things are easier to get done, such as scalability, fault tolerance and recovery. However, this does not mean that things happen automatically. Things must be defined and they must be taken into account in planning.

Perhaps the most convincing of Azure’s services was Cosmos DB and the possibilities it offers for maintaining databases in a distributed manner. It was reassuring to see how consistent the databases were and how quickly they synced. Correspondingly, it was bad that the user was offered only graphical representations of the state of the databases. It would have been more interesting to look at some kind of logs of the events.

For someone who has not used PaaS services offered by Azure or others, it may come as a surprise how much work it actually requires to implement various components and define their functionalities. Another thing that requires time is to implement and plan the transmission of messages between these components in use.

Of course, PaaS services are not for every purpose, but for our style of implementation it is optimal. Due to lack of time, we did not implement the Azure Front Door service. With this, we could also have distributed the front side globally. With the help of this and a few other components, we would have had a really well distributed, consistent and fault-tolerant system.

At the first lecture we were told “Use distribution only when you cannot avoid it”. Based on this project, we dare to add to that. If you really have to, find out what kind of possibilities PaaS gives you and whether it is suitable for your purpose.

For someone already familiar with PaaS and Azure, but who had not done any software architecting in a while, the project provided a good context in which to compare different services and to try and select the best fits based on the requirements. Event-driven architectures seem to be a hot topic right now in public cloud projects, and it’s rather great opportunity to really think about what of the requirements need to implemented with code, and which can be done just by configuring the available services.

## Group member participation

We roughly formed up like this:

*Initial design: Jukka and Jari.*

*Architectural design: Jukka (lead), Jari, Ville.*

*Frontend SPA implementation: Jari*

*Frontend function implementation: Jari, Jukka*

*IAC implementation: Jukka*

*Service Worker Implementation: Jukka, Ville, Jari.*

*DbApi implementation: Ville*

*PaaS service configuration: Jukka*

*Documentation: Jukka, Jari, Ville.*

For communication we used a Telegram message group. We mostly worked remote, with everyone meeting up two times in study groups. Every member of the group works alongside studying, so we initially agreed that everyone participates based on available free time, and most of the group members had to go on work- and other trips during work. Based on this, we feel that it’s fair to share points evenly among members.