**Case study report: Shaping & Building your initial Architecture**

**PREPARED FOR**

Frank Coenen

**PREPARED BY**

Maarten Blömer

Nick Krijgsman

Faruk Aydin

Jursley Gonzalez

Vincent Andersen

# **Table of Contents**

[**Table of Contents**](#_pykfpgqrlej) **2**

[**Case Study**](#_3s39dys3cx66) **5**

[Case goal:](#_qwqn8mh5s18) 5

[Case questions:](#_4frtw1206ej) 5

[Questions:](#_rhg0mb87rhxw) 5

[**Requirements**](#_tzn1rejxiu56) **6**

[**Brainstorm**](#_whir32t8ue2v) **7**

[**Individual Research**](#_wisaxg2eeii0) **8**

[Web-Queue-Worker (Nick)](#_r6mbpycho6yj) 9

[Ease of transition](#_mmbwgpt8ktx9) 9

[Pros & Cons](#_hlt0xnz95kyy) 9

[Pros](#_pet25day2im6) 9

[Cons](#_251ayooikmh9) 10

[Scalability](#_tqfbi1y2azpv) 10

[Maintainability](#_2xn2hen3kqkk) 10

[Performance](#_k01dnx310xvr) 10

[Microservices (Vincent)](#_y4cfri41hmke) 11

[Ease of transition](#_879hwkgmaf50) 11

[Initial steps](#_n4fwzpvh3d2x) 11

[Verdict](#_udf2ckdcp5gj) 12

[Pros & Cons](#_bozvti8tlf03) 12

[Pros](#_5e1qexb884gi) 12

[Cons](#_6y2klqmpz3b3) 12

[Scalability](#_niw64foj5qhw) 13

[Maintainability](#_ou0im12u9tis) 13

[Performance](#_x7pmbkvi97c0) 13

[Event-driven architecture (Jursley)](#_pvtuy1cps5p0) 14

[Ease of transition](#_khg8hgqo42co) 14

[Pros & Cons](#_ri2h7m5mrpmo) 14

[Pros](#_7r4na6sfm7ha) 14

[Cons](#_4k0wdzlx081z) 14

[Scalability](#_jofz2sw5899) 15

[Maintainability](#_n6k5qz8lijoy) 15

[Performance](#_lex2b0z492yd) 15

[Big data - Maarten](#_ulpocucymcot) 16

[Ease of transition](#_d3ou08u2sic5) 16

[Pros & Cons](#_t5ri3ncc9lr0) 16

[Scalability](#_d1f274js8d51) 17

[Maintainability](#_bqitv4y3iiht) 17

[Performance](#_c48grn70rea7) 17

[Big compute (Faruk)](#_dmpgavat4khz) 18

[Ease of transition](#_4rzv3ucgc686) 18

[Pros & Cons](#_nm7ysm6g7819) 18

[Scalability](#_eddsycvf09az) 19

[Maintainability](#_8lvo98fhd72b) 19

[Performance](#_8ylxt3f6mg2n) 19

[**Matrix**](#_9awyqziu69mk) **20**

[**Conclusion**](#_ortrbpg268ws) **21**

# Case Study

## **Case goal:**

* Investigate the process of defining an initial software architecture that fits your customer needs (functional and non-functional) and the technical limitations.
* Know several architectural styles which are scalable, maintainable and can handle a large number of service requests. Secondly, know the pros and cons of these architectural styles.
* Explore how to extend the initial high level architecture to a more technical architecture that matches the needs of your customers and the most important non-functional requirements.

## Case questions**:**

### Questions**:**

* Ease of transition
* Pros & Cons
* Scalability
* Maintainability
* Performance

# Requirements

## 

|  |  |
| --- | --- |
| **Type** | **Requirement** |
| **FR.01** | Magazine distribution 4times a year |
| **FR.02** | Ticket Sales |
| **FR.03** | Refunds |
| **FR.04** | Reservation of train tickets |
| **FR.05** | Reservation of hotel rooms |
| **FR.06** | Reservation of bicycles |
| **FR.07** | NS customer panel |
| **FR.08** | Support international travels |
| **FR.09** | Face recognition/biometrics support |
|  |  |
| **NFR.01** | Offered services should be reliable |
| **NFR.02** | Performance should automatically scale when requests for specific services increase |
| **NFR.03** | It should be possible to develop services and manage services runtime independently. |
| **NFR.04** | Phasing out old functionality and introducing new services gradually, without major interruptions of the system |
| **NFR.05** | Serverless, work with cloud solutions |

# 

# **Brainstorm**

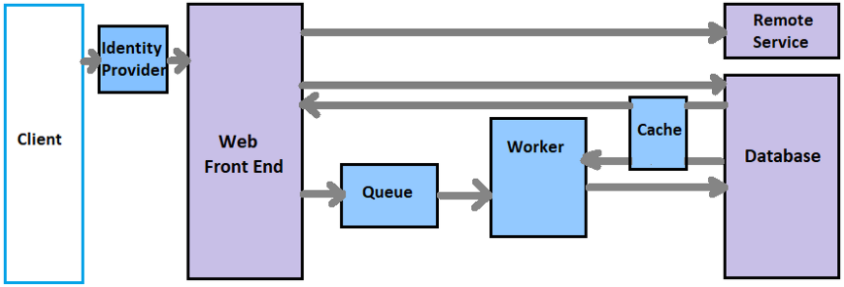
# **Individual Research**

## 

|  |  |
| --- | --- |
| **Architecture** | **Person** |
| Web-Queue-Worker | Nick |
| Microservices | Vincent |
| Event-driven architecture | Jursley |
| Big data | Maarten |
| Big compute | Faruk |

# 

## Web-Queue-Worker (Nick)



The application has a web front end that handles HTTP requests and a back-end worker that performs long running CPU-intensive tasks in this style. The front end communicates to the worker through an asynchronous message queue. The web and worker are both stateless. Session state can be stored in a distributed cache. Any long-running work is done asynchronously by the worker. Any short read/write functions can be done directly into the database without issuing the worker. The worker is specifically for large operations that might take a while to complete.

### Ease of transition

I couldn’t find a lot of information about transitioning from a monolithic architecture to a Web-Queue-Worker based architecture. The most important thing to do is to install a worker and a queue between your front end and your backend. The next step is to categorize your calls. Which calls are quick and which are time consuming. The quicker calls can be sent to the backend directly. The longer more time consuming calls have to be sent to the queue and the web worker will execute the messages in the queue.

### Pros & Cons

#### Pros

* Relatively simple architecture that is easy to understand.
* Easy to deploy and manage.
* Clear separation of concerns.
* The front end is decoupled from the worker using asynchronous messaging.
* The front end and the worker can be scaled independently.

#### Cons

* Without careful design, the front end and the worker can become large, monolithic components that are difficult to maintain and update.
* There may be hidden dependencies, if the front end and worker share data schemas or code modules.

### Scalability

By using the Web-Queue-Worker architecture the frontend and backend are decoupled by definition. This makes it so that it is easy to scale the different components when needed.

### Maintainability

A Web-Queue-Worker architecture has a few main componenten, one being the frontend and a second one being the worker/backend. For maintenance this means it is quite easy to understand the architecture is easy to understand and can be updated/fixed without needing a lot of knowledge about the architecture.

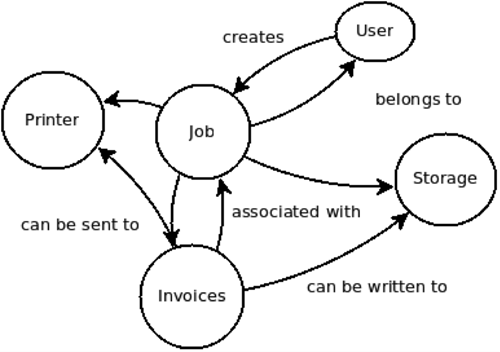
### Performance

The Web-Queue-Worker architecture is best suited for large operations that take a while to complete. Due to handling the request asynchrone it doesn’t matter if a time consuming job just started and a smaller job having to wait for it to complete.

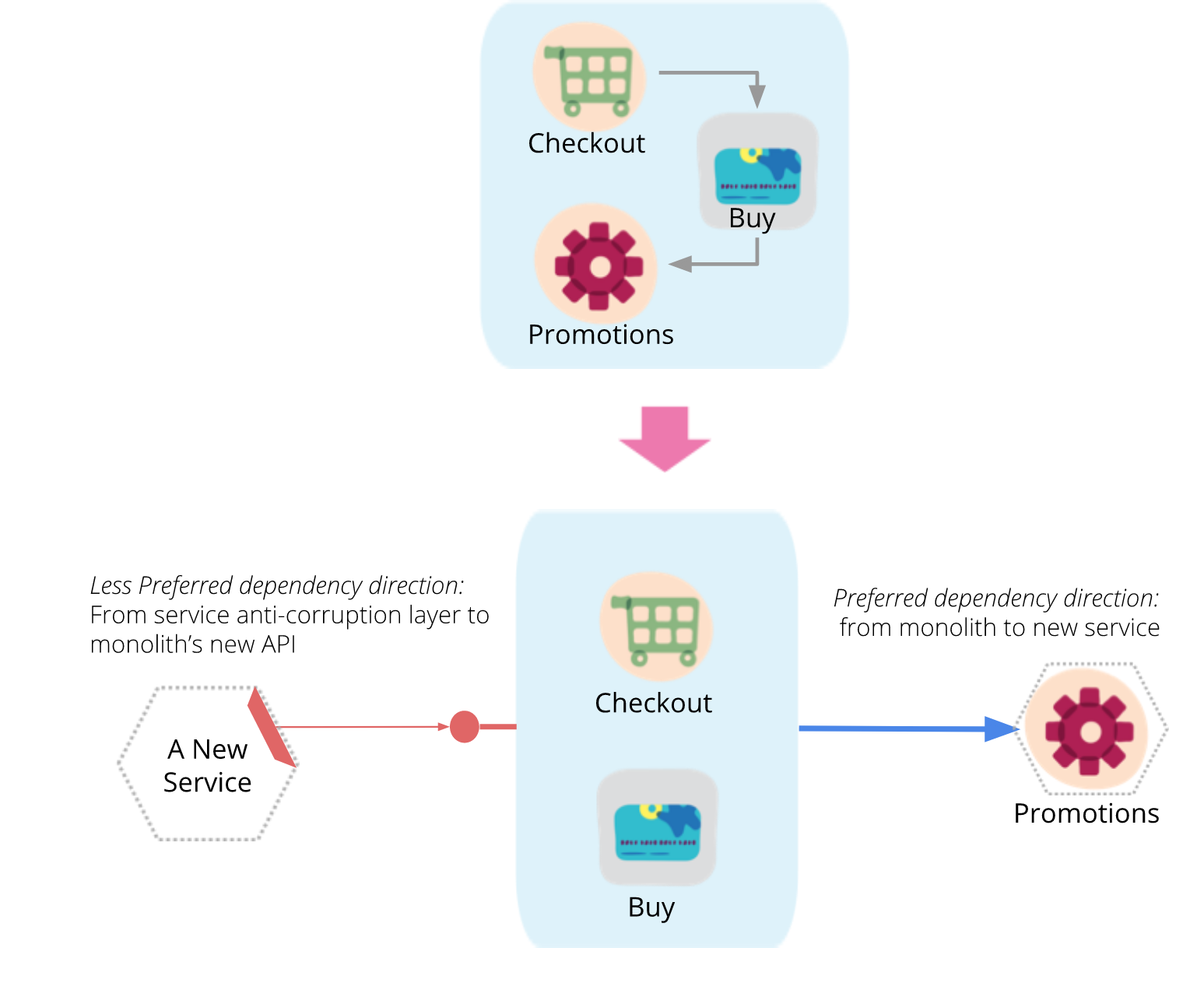
## Microservices (Vincent)

### Ease of transition

#### Initial steps

Monolith applications that are looking to transition into microservices should look at sections of their application that are already fairly decoupled and/or have a low dependency of other parts of the system. This can be visualized by drawing an application graph within the context of the application. The parts that will find themselves on the edge of the graph are often parts that can be easily removed and abstracted away from the application.  


By starting with the edge services, developers will get some practice on decoupling and things like monitoring. This will also mean that they won’t be able to break crucial parts of the system. When building their new microservices the developers should focus on minimizing dependencies from the microservice to the monolith and instead implement it the other way around.

*  
Decouple the service that doesn’t require a dependency back to the monolith first and minimize changes to the monolit*h  
  
When exploring deeper into the monolith architecture, a lot of components may inherit multiple functionalities which need to be carefully thought about and split into multiple different microservices. Identify the most coupling concept and decouple, deconstruct and conceptualize into concrete domain services.

#### Verdict

From here on out, decoupling becomes more case specific as to what the developers are able to decouple regarding read/write from databases or other functionalities from the monolith. Guides exist to decouple a monolith into microservices such as [Martin Fowler’s](https://martinfowler.com/articles/break-monolith-into-microservices.html) take on it which is briefly described above. In terms of ease, there are certain guidelines that developers are able to follow which help a lot. Transitioning from a monolithic architecture to a decoupled architecture will always come with challenges but when done correctly will serve to be more useful in the long run.

### Pros & Cons

#### Pros

* Independent development and deployment
* Speed of development on single microservice
* Usage of multiple code languages per service
* Easily modifiable
* Faster deployment
* Better fault isolation
* Data isolation
* Easy to scale
* No need for long term commitment to technology stack
* Easy to implement in CI/CD pipelines (single repo)
* Cloud-readiness

#### Cons

* Writing tests takes up more time
* Lots of microservices increases the barrier of entry
* Integration of different systems can be complicated on a larger scale
* Extra effort has to be made in a messaging system
* There is a higher chance of failure during communication between different services
* More complexity in communication between different systems
* Needs more collaboration between teams

### 

### Scalability

Microservices are independent from each other which also means that different parts or microservices within an application can be scaled up or down depending on the user load. E.g. in a webshop the catalogue service may be scaled up independently from the rest as more users browse through the items while the purchasing service will only be scaled up a bit. Scalability is one of this architecture's biggest strengths as it has lots of tools and frameworks that supplement this ideal.

### Maintainability

The microservice architecture tackles maintainability in a great way by having decoupled services that together, form the application. This means that the services on their own are less dependent on one another which means that new functionalities in the future can be added more easily to different parts of the system. This also means that services can be independently tweaked or removed since doing so won’t break the whole application.

## Performance

Since microservices are easily scalable the performance is great when met with a high volume of user requests. The system should be able to hold on its own and be scaled accordingly to meet the performance it needs to deliver.

## Event-driven architecture (Jursley)

### Ease of transition

The event-driven architecture pattern is a relatively complex pattern to implement, primarily due to its asynchronous distributed nature. When implementing this pattern, you must address various distributed architecture issues, such as remote process availability, lack of responsiveness, and broker reconnection logic in the event of a broker or mediator failure.

One consideration to take into account when choosing this architecture pattern is the lack of atomic transactions for a single business process. Because event processor components are highly decoupled and distributed, it is very difficult to maintain a transactional unit of work across them. For this reason, when designing your application using this pattern, you must continuously think about which events can and can’t run independently and plan the granularity of your event processors accordingly.

Perhaps one of the most difficult aspects of the event-driven architecture pattern is the creation, maintenance, and control of the event-processor component contracts. Each event usually has a specific contract signed to it (e.g., the data values and data format being passed to the event processor). It is vitally important when using this pattern to settle on a standard data format (e.g., XML, JSON, Java Object, etc.) and establish a contract versioning policy right from the start.

### Pros & Cons

#### Pros

* Loosely coupled structures
* Complete isolation of the microservices
* No synchronous REST calls
* Asynchronous event-driven functionality
* Performance gain
* Messages are identical to all subscribers
* Self describing payloads (JSON or XML)

#### Cons

* If your messagebus system encounters production failure the whole system will fail.

### Scalability

EDA can be used on its own or to support a microservices architecture. Its scalability and resilience benefits are especially valuable supporting a microservices architecture.

A microservice is a service that’s tightly scoped, loosely coupled, independently deployable, and independently scalable. Because they’re tightly scoped and loosely coupled, they’re easier to manage as a product with a standing team for faster delivery. You also get greater agility, since smaller deployable code packages result in smaller regression testing. Smaller units running on private cloud containers have more scalability, allowing you to scale up hardware for the one service without having to scale up a giant monolith.

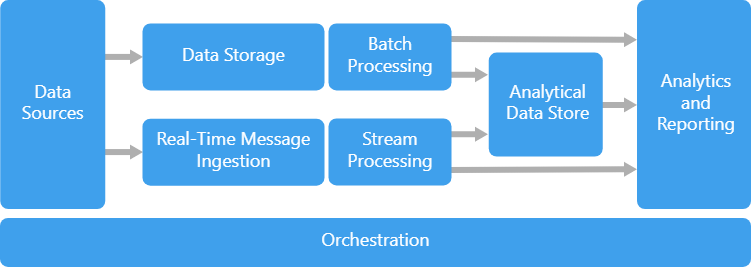
### Maintainability

### Performance

With EDA, events do not require a reply and are inherently asynchronous, which means events can be consumed and processed later, if a service is busy or unavailable. In addition, services are all decoupled in the EDA model, which means that if a service goes down it does not impact any of the other services in the chain. In this way, EDA offers high resilience to failure, assuring vital business continuity

### 

## Big data - Maarten



For big data structure to work it requires one or more of the following; data sources, data storage, batch processing, real-time message ingestion, stream processing, analytical data store, analysis and reporting, orchestration. It is advised to only use this architecture when you have too much data to store and process in volumes that are too large for traditional databases, transform unstructured data for analysis and reporting, Capture, process and analyze unbounded streams of data in real time, or with low latency.[[1]](#footnote-0)

### Ease of transition

The big data architecture is very similar to a monolithic structure. The main difference is that there is an additional layer of analysis surrounding it. Transitioning into a different architecture would require to remove all these layers.

### Pros & Cons

Pros:

* Scales well with lots and lots of data
* Productivity
  + There are lots of tools that help analyse the data being exchanged at a fast pace.

Cons:

* Complexity
* Security
  + With everything stored in one place it becomes very vulnerable
* Costs
  + Lots of server space costs lots of money
* Quality
  + It’s crucial to have people working on the accuracy of the data that’s being collected and stored. High risk that the data isn’t accurate. This inevitably leads to higher costs.

### Scalability

Big data is easy to scale into an even bigger data collection system. The way it’s structured already takes into account lots and lots of data.

### Maintainability

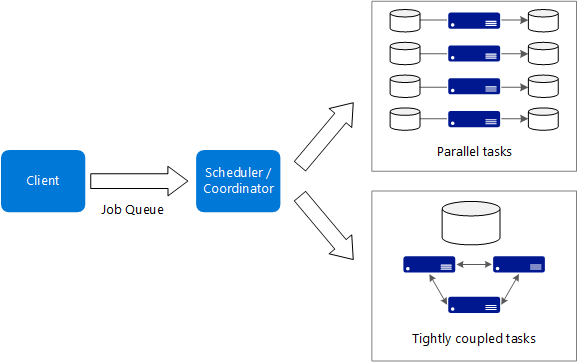
Difficult to maintain as everything is stored in one big space. However it is easy to find the data you require as everything is analysed. There are some pitfalls when it comes to this as the data collected might sometimes not be accurate.

### Performance

Based on the fact that everything requires a lot of computing power to analyse the data and to access it the performance isn’t much better than a standard monolithic architecture.

### 

## Big compute (Faruk)



### Ease of transition

### Pros & Cons

Pros:

* High performance with "embarrassingly parallel" processing.
* Can harness hundreds or thousands of computer cores to solve large problems faster.
* Access to specialized high-performance hardware, with dedicated high-speed InfiniBand networks.
* You can provision VMs as needed to do work, and then tear them down.

Cons:

* Managing the VM infrastructure.
* Managing the volume of number crunching
* Provisioning thousands of cores in a timely manner.
* For tightly coupled tasks, adding more cores can have diminishing returns. You may need to experiment to find the optimum number of cores.

### Scalability

Big Compute shines at large scales. Easily parallelizable workloads are the best use cases, because they break the workload into independent tasks. This is where we can gain the most from large numbers of compute cores. Big Compute is all about executing any software package, written in any language by passing in variables. This creates an opportunity for developers to optimize their code to be efficient.

### Maintainability

It depends on how the developer creates the architecture. If there are a lot of tight couples, then it will be hard to maintain. Otherwise it will be more efficient to maintain.

### Performance

It performs at it’s best on large scale platforms. For lower scale projects it’s recommended to use another architecture type. This type of architecture is suited for:

* Computationally intensive operations such as simulation and number crunching.
* Simulations that are computationally intensive and must be split across CPUs in multiple computers (10-1000s).
* Simulations that require too much memory for one computer, and must be split across multiple computers.
* Long-running computations that would take too long to complete on a single computer.

# Matrix

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Pugh Matrix** | **Weight** | **Web-Queue-Worker** | **Microservices** | **Event-driven architecture** | **Big data** | **Big compute** |
| **Ease of transition** | 5 | + | - | - | - | ? |
| **Scalability** | 5 | + | + | + | + | + |
| **Maintainability** | 5 | + | + | + | + | -/+ |
| **Performance** | 4 | - | + | + | - | + |
|  | total + | 15 | 14 | 14 | 10 | 11 |
|  | total - | 4 | 5 | 5 | 9 | 2 |
|  | total | 11 | 9 | 9 | 1 | 9 |

# Conclusion

|  |  |
| --- | --- |
| **Question** | Investigate the process of defining an initial software architecture that fits your customer needs (functional and non-functional) and the technical limitations. |
| **Question** | Know several architectural styles which are scalable, maintainable and can handle a large number of service requests. Secondly, know the pros and cons of these architectural styles. |

The process of defining an initial software architecture is highly dependent on what the customer needs are. Our approach to the case study situation was defining the needs of the customer and along with which architectural points are relevant for the case. For that, we have chosen the following criteria to grade the different available architectures on:

* Ease of transition
* Scalability
* Maintainability
* Performance

While most of these points are obvious to get into when choosing a new architecture style, the case came with a non negotiable requirement which was the graduality in implementation of the new architecture. This is where the ease of transition comes in which defines the possibility and/or ease the developers face when converting their monolithic architecture, all the while keeping the main system stable.

The architectures we have chosen were based on a blog covering Architecture Styles by [Microsoft](https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/). We could have been more critical of our architecture choices by looking at e.g. :

* most commonly used architectures
* architectures that have been used in the past that deal with the problems that came forth within our case
* known architecture styles within our team

But since we did not want to spend too much time on researching architectures, we have simply chosen the ones that were easily accessible to us. Because of this, we have also made the mistake of simply choosing different *architecture styles* while not paying heed to the scalability of multiple different styles and choosing from those.   
To avoid this mistake in the future we could have tried to comprehend the question on our own and shared our interpretations from the question with each other.

# 

|  |  |
| --- | --- |
| **Question** | Explore how to extend the initial high level architecture to a more technical architecture that matches the needs of your customers and the most important non-functional requirements. |

The needs of our customer (NS) can best be handled in a microservice architecture. This is because of the large amount of different services the NS wants to offer its customers. By using a microservice architecture certain services are scaled more easily. Another benefit is that the system can gradually be built. Even for future services the NS wants to implement, these can be developed in a microservice and eventually added to the system. To have a clear overview for the client (NS) and for the developers it is a good practise to have a bit more technical architecture instead of using a high level architecture that will just state which microservices exist. To give a more hands on example making and discussing a C2 model with the client can be helpful for both the client and the developer to get a better understanding of the client’s wishes and demands.

1. <https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/big-data#:~:text=A%20big%20data%20architecture%20is,big%20data%20sources%20at%20rest>. [↑](#footnote-ref-0)