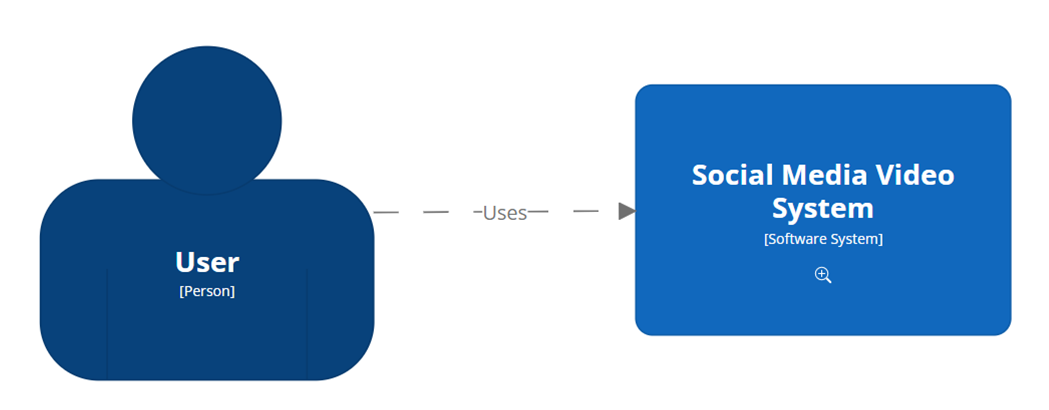
Engineering 2 Assessment

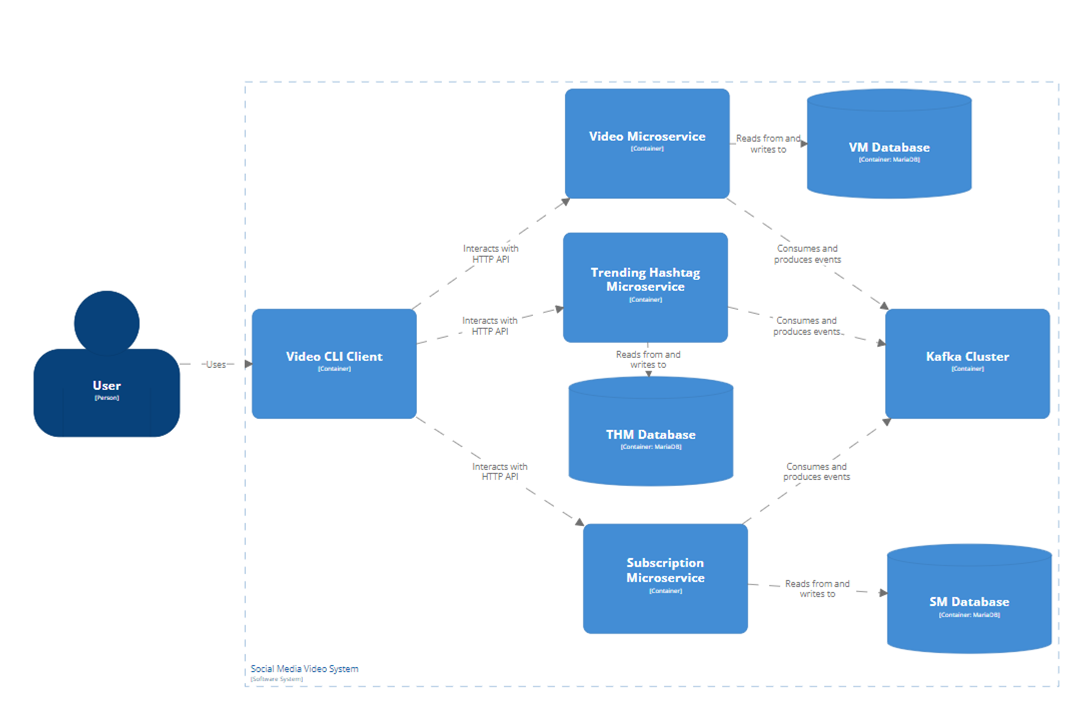
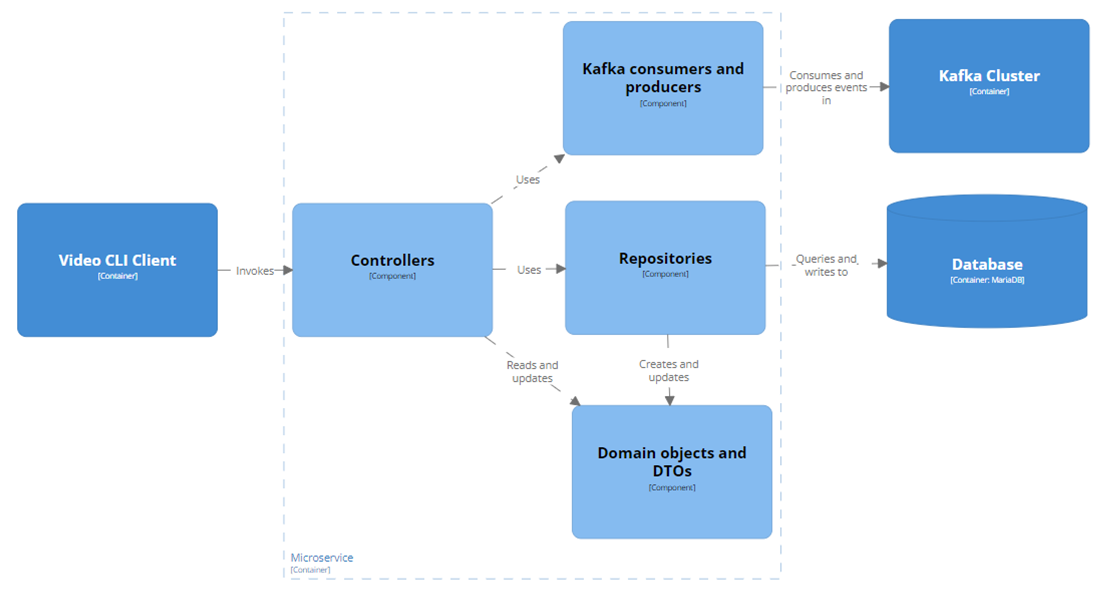
2.1.1 Architecture

Below are the C4 diagrams and different abstraction levels:

Context:



This shows the user interacting with the system

Container: Component: 

One way that this design scales with increasing user demands is that each microservice can be deployed separately on individual nodes and so requests from the user (through the CLI Client) are spread to the different nodes resulting in less load per node. However, this may be a marginal gain as the microservices are unlikely to have an equal amount of requests. This can be counteracted by adding spinning up more nodes with a copy of the more frequently used microservices. The bottleneck would then be the communication between the microservices of the same type (something that hasn’t been implemented) as they would need to make sure the databases are kept up to date with any changes.

A good feature of this design is that new requirements can be implemented with relative ease as a new microservice can be created to facilitate this new requirement without needing to change anything about the other microservices. This makes it much less prone to faults and also easier to implement without too much knowledge of the current system.

2.1.2 Microservices

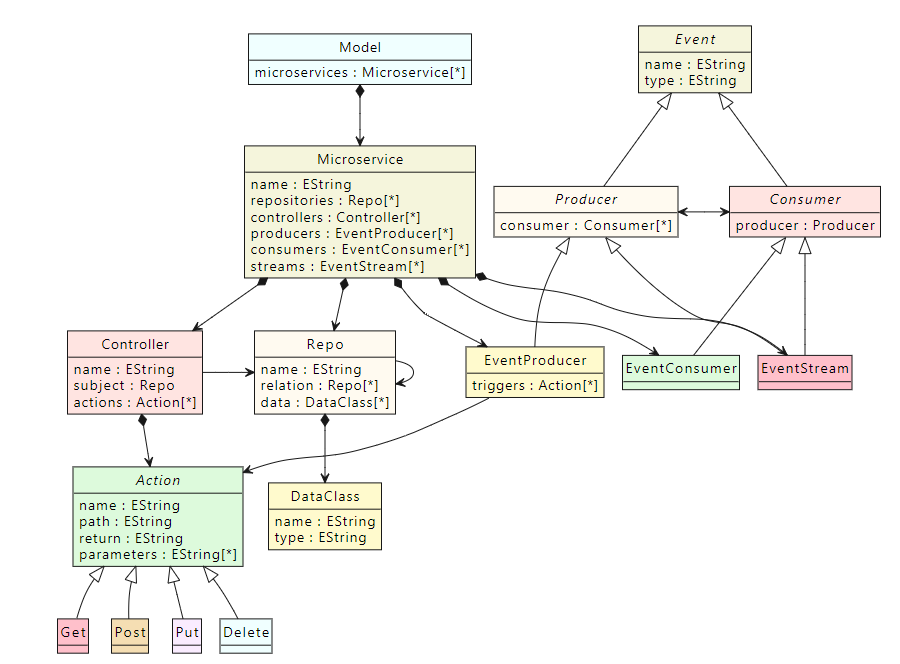
2.1.3 Containerisation

One way that this solution can scale up larger numbers of users, is by increasing the resources allocated to containers that need them. This may only help marginally as the main bottleneck could be caused by the sheer volume of requests rather then how expensive each request is. This can be counteracted by adding more containers which can share the request load, decreasing the individual load. It can be resilient to failures partially by having a separation of functionality meaning that if all the containers or nodes that are used for some part of the functionality, then users can still use the other parts. Having multiple containers for the same functionality will also help as if one fails then the other can pick up the requests that would have been directed to the failed one. Another way it is resilient is that by restarting the nodes that fail. It minimises downtime and so while it has failed, the impact to the user experience will be less.

2.1.4 Quality Assurance

2.2.1 Metamodel

This is my metamodel:



This design was mostly guided by a mix of: the typical structure of this kind of microservice, the criteria for what must be included, what would make automated generation easier and how easily a none-domain expert would be able to use it. While typically a domain expert would be making the model, I decided to try and make the metamodel accessible as this allows experts to present and explain their model to others such as developers or other stakeholders, something that is important in a productive work environment.

The first design choice I discounted was having the Microservice as the top-level class rather than the Model. I initially liked the idea of modelling each microservice separately as separation of concerns is generally a very good design principle. However, this would limit the constraints I could put on it and more importantly disable the ability to view the event stream connections which would have made it harder to develop and present a group of microservices. While I haven’t stated any positive of my decision but rather only negatives of the alterative, there are not any perfect solutions and so often it is better to opt for the ‘least worst’ solution.

An interesting design choice I made is the inheritance of the events. It does look needlessly complex, something I really tried to avoid, however I argue that: when making the model it isn’t any more complex than having no inheritance, when viewing a model graphical syntax, it is more up to the syntax design to including any complexity and when presenting or explaining a model again if anything it helps as it allows you to give an appropriate level of abstraction. I chose this design mainly for the ease to have producers reference consumers and consumers reference produces, something that could have been difficult when you have a type that is both a producer and consumer. Having them inherit from the Event class as well doesn’t have much functional impact as the Event attributes could just be inserted into each derived concrete class, however it does enable new views in the graphical syntax should a designer want. The Microservice class could have contained all the concrete classes as Events, however this was discounted as it makes designing code generation more complex and it doesn’t allow for a more heavily implied separation of concerns.

It was stated that modelling a microservice domain is not necessary for this assessment, and while this may be the case, I argue that a microservice (in the context of the ones we are creating for this assessment) only contain 3 things: a database (the domain), a public interface (the controller) and a private interface (the events). While all 3 of these can be optional and a microservice can function with only 2 of these, I would say that they are the quintessential building blocks of a microservice and so not modelling 1 of these would be under representative at the least and could be potentially damaging. This would also leave it database design up to the developer when it’s more in the architect’s responsibility.

I initially wanted to model the domain further and include more detail in the Repository - Repository relation. I would have added the type of relation (such as ManyToOne and containment) as a lot of difficulties when developing the microservice were from domain issues. However even though I said it should be the architect’s responsibility I decided to leave it more to the developer as I was conscious about modelling too much of the microservice or overcomplicating it which relation classes can often do.

The Action class (HTTP methods) have the attributes they do either because it made it easier to generate code and extend any generated code or because it required as part of the assessment. Action could have been a concrete class and have the different HTTP method types as an Enum that Action has as an attribute, however having it this way so that the methods are concrete types that inherit from action makes it easier for to view separately in a graphical syntax should it be designed that way for little if any change in complexity.

Another thing worth mentioning is that the EventProducer – Action relation has no impact on generated code, it purpose is to inform developers of where event should be triggered from and so that it can be shown in a graphical syntax to highlight the data flow.