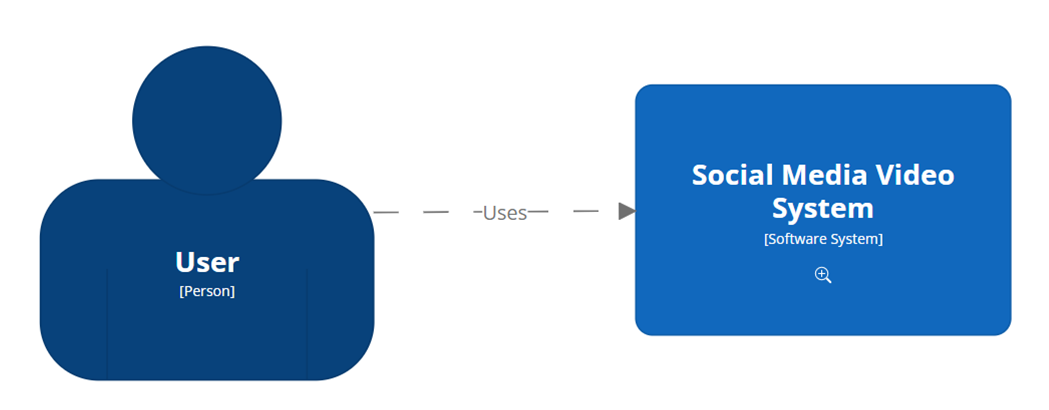
Engineering 2 Assessment

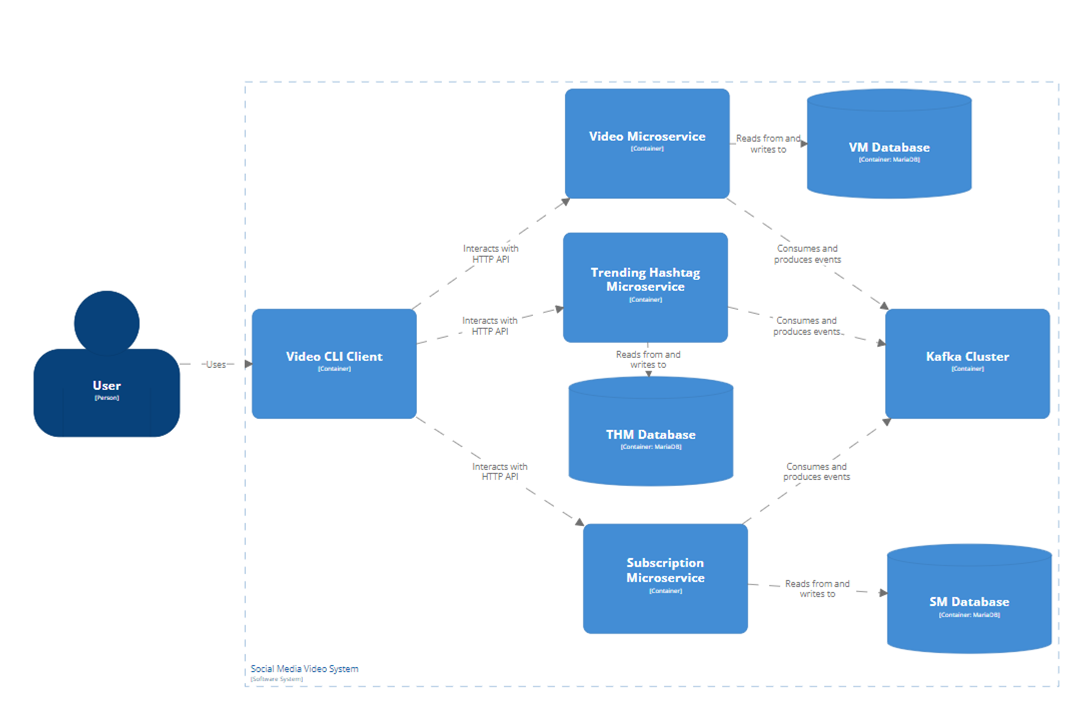
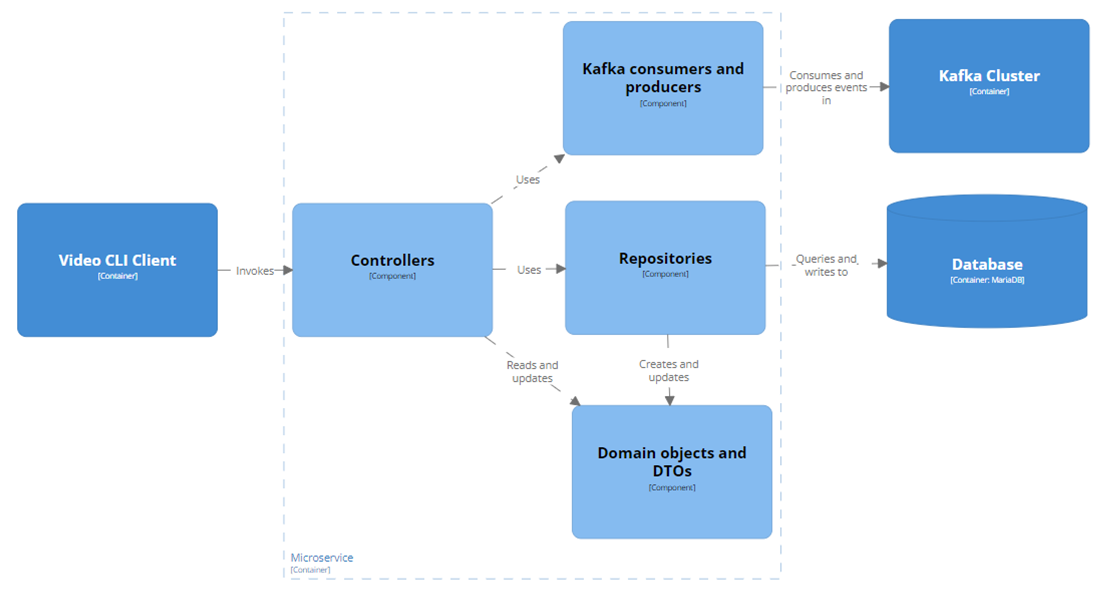
2.1.1 Architecture

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

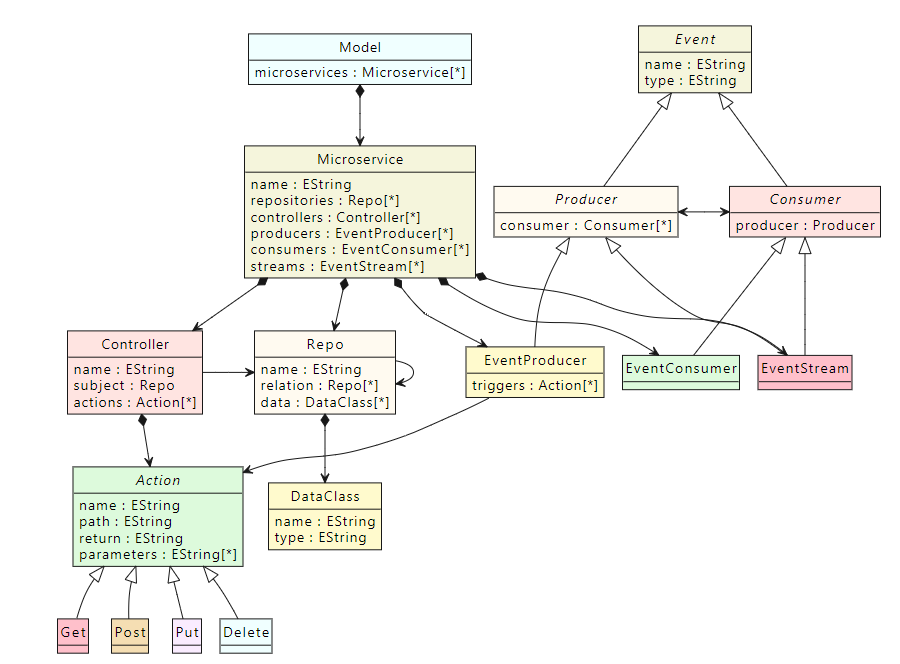
2.1.3 Containerisation

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:



The 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,