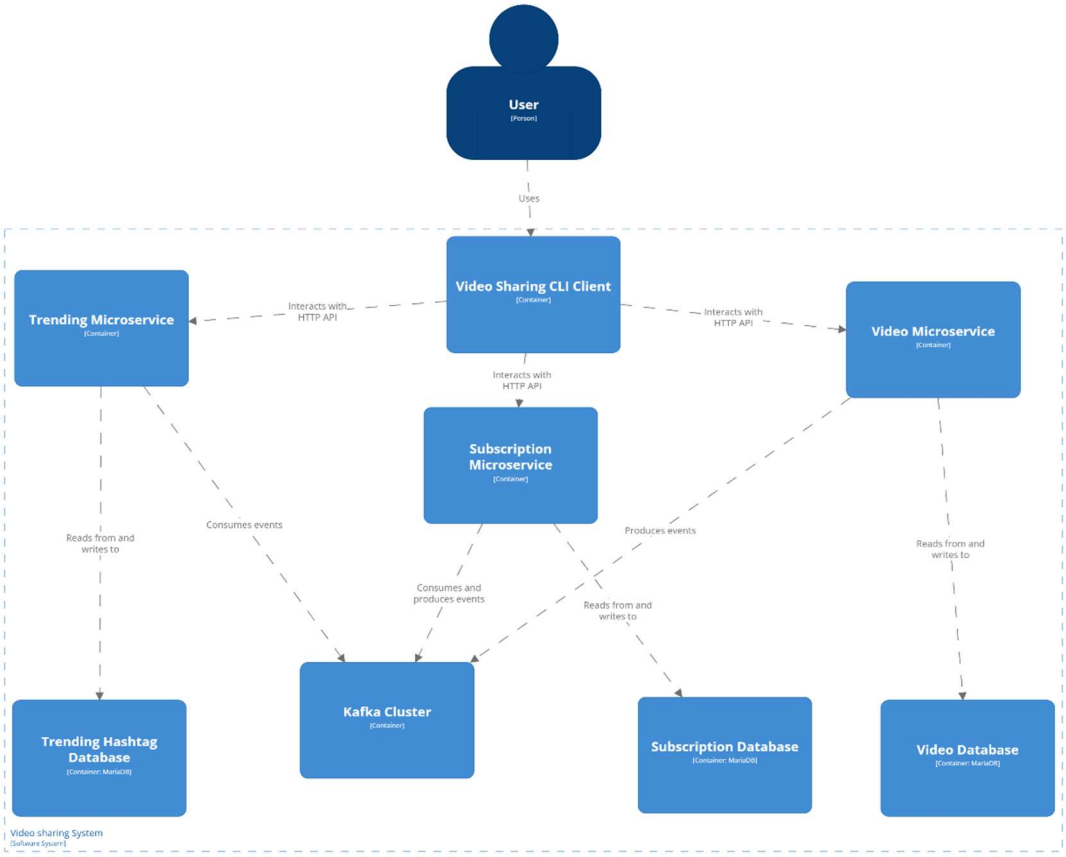
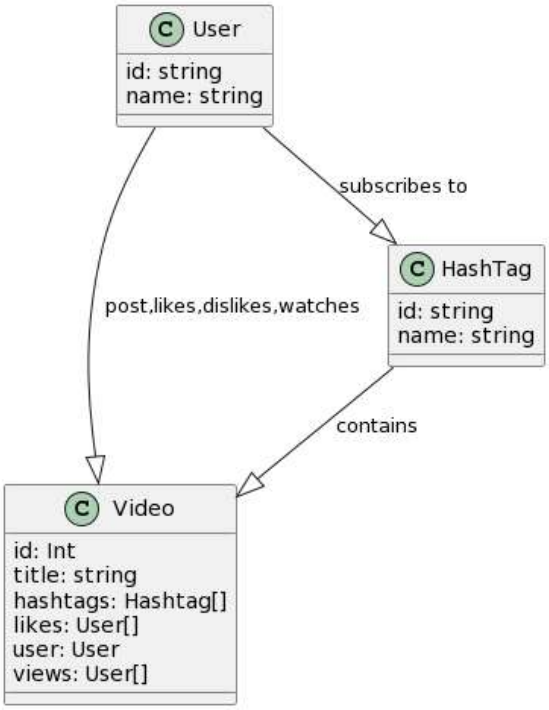
Engineering 2: Automated Software Engineering

Exam number: Y3889274

2.1 Data-Intensive Systems

2.1.1 Architecture





By storing the videos likes and dislikes as a set of users rather than an integer, we can scale up the system to recommend videos based on which videos a user has liked or disliked. This could also assist in recommending videos with specific hashtags. As well as subscribing to a hashtag, we could recommend videos based on which hashtags are assigned to a liked video.

Using a microservice architecture allows us to scale with increasing user demands as it allows different services to be scaled depending on its individual needs and allows us to independently split up different services, should they become too big, to handle each specific business capability. For example, within the video service, as the functionality of a video changes, e.g. a high volume of videos to store, we could scale this independently to the rest of the services. We could also remove the management of a user to a new user microservice.

The use of communication using events allows for a complete decoupling of the microservices so as we scale and separate the microservices, communication between the microservices will not be affected as they are not communicating directly with each other. Also, the use of events means we could use asynchronous events as the system is scaled as asynchronous communication allows for message queues so as traffic in the system increases, we can prioritize, and store messages prevent overloading the system.

.

2.1.2 Microservices

The system is split into three microservices: the video-microservice, trending-microservice and the subscription-microservice. These services all communicate with each other through Kafka events.

The video microservice handles the management of videos, users, and hashtags. It allows users to create, view, like and dislike videos. Within the CLI, it contains 6 commands: post-video, watch-video, like-video, dislike-video, list-videos, and list-videos-by-user.

The post-video command takes the parameters: a title, a list of hashtags and a username. It then checks the user database for an existing user with the given username. If there is not a user created with the username, it will create a user, save it to the service's user database and assign it to the new video as the poster. Given the hashtags in the request, which is a string of a comma-separated list, the post-video endpoint splits the string into the individual hashtags, and similarly to the username search, checks for a hashtag and creates one if there is not an existing hashtag, saving it to the service's hashtag database, and adds it the videos hashtags. It then saves the video to the video database and publishes an event sending the new video-id and video. This event is then subscribed to by the trending and subscription service.

The decision to use a username rather than the user id in the request was made to encourage user friendly behaviour as it is easier to remember a username rather than an arbitrary number. For this reason, all endpoints that require a user, requests a username.

The video microservice also has the like-video, dislike-video, and watch-video request. All three of these commands take a video id and a username in the request. Similarly, to the post-video command, these commands check the user database for a user matching the given username and creates one if there is not an existing user, then adds the user to the video's likes, dislikes, or views accordingly. All three of these endpoints publish an event with the video and user objects: the like-video event is consumed by the trending service and the watch-video event is consumed by the subscription service.

The list-video command takes no request parameters and returns the full list of videos posted. The list-video-by-user command takes a username and returns a 404 error message if the user is not found, otherwise, it returns the list of videos posted by the user.

The trending service subscribes to the post-video event and consumes these messages to generate a database of hashtags. The trending service also subscribes to the `like-video` event, consuming these messages to stream the trending hashtags in a rolling window of an hour.

The subscription service allows the user to subscribe to hashtags and list the next ten videos to watch. The service consumes the `post-video` and `watch-video` events from the video-microservice to create its database of videos, it then uses the watch-video event to determine the most popular videos. The `list-videos-to-watch` endpoint takes in a username and a hashtag and lists the most popular videos with that hashtag that the user has not already seen. The subscription service also has the endpoints `subscribe-to-hashtag` and `unsubscribe-to-hashtag` that given a hashtag and username, allow a user to subscribe and unsubscribe to a hashtag.

For the full list and usage of the commands, please refer to the ReadMe.md file within the top-level `microservices` directory.

2.1.3 Containerisation

Each service has its own docker container and within the docker-compose file we declare the number of clusters and partitions within the Kafka clusters. This allows for scaling as we increase the number of services, we can just increase the number of declarations of docker containers for the services. The use of docker-compose allows us to define the number of partitions of the Kafka clusters, so as the number of messages sent between services increases, we can increase the number of the partitions.

Docker also allows us to configure health checks on individual container so if a container fails or becomes unresponsive, the orchestration tool can automatically replace it with a healthy instance. This ensures resilience to failures and minimizes downtime.

Deploying the system in containers also allow for horizontal scaling by replicating the containers across multiple hosts, so as traffic increases on the system, the new instances of the application can be created to handle the load

2.1.4 Quality Assurance

Tests

Within the video microservice, I have tested each controller method. Unfortunately, these tests did fail with a timeout error as the controller methods waited for events to be sent. To enable theses tests to pass I would need to mock my Kafka producers, so that we would not be sending actual events.

Security vulnerabilities

When inspecting my docker image for vulnerabilities, I found the highest number of them within the MariaDB image with 36 vulnerabilities. The video-microservice image had the second highest number with 29 total vulnerabilities including 4 critical vulnerabilities and 18 high-level vulnerabilities. I reduced this to only two high-level vulnerabilities by updating the version of alpine used to build the image.

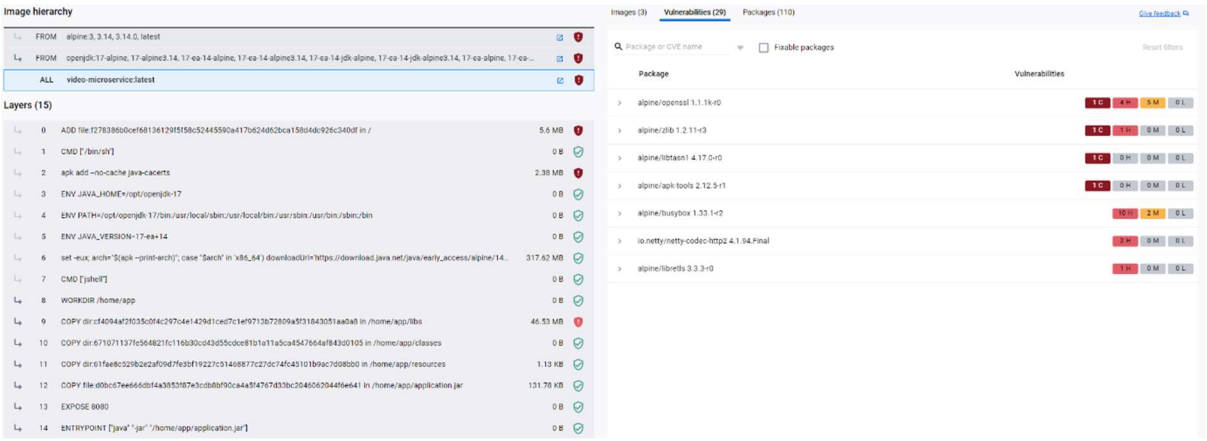


Figure 1: video-microservice image vulnerabilities prior to fix

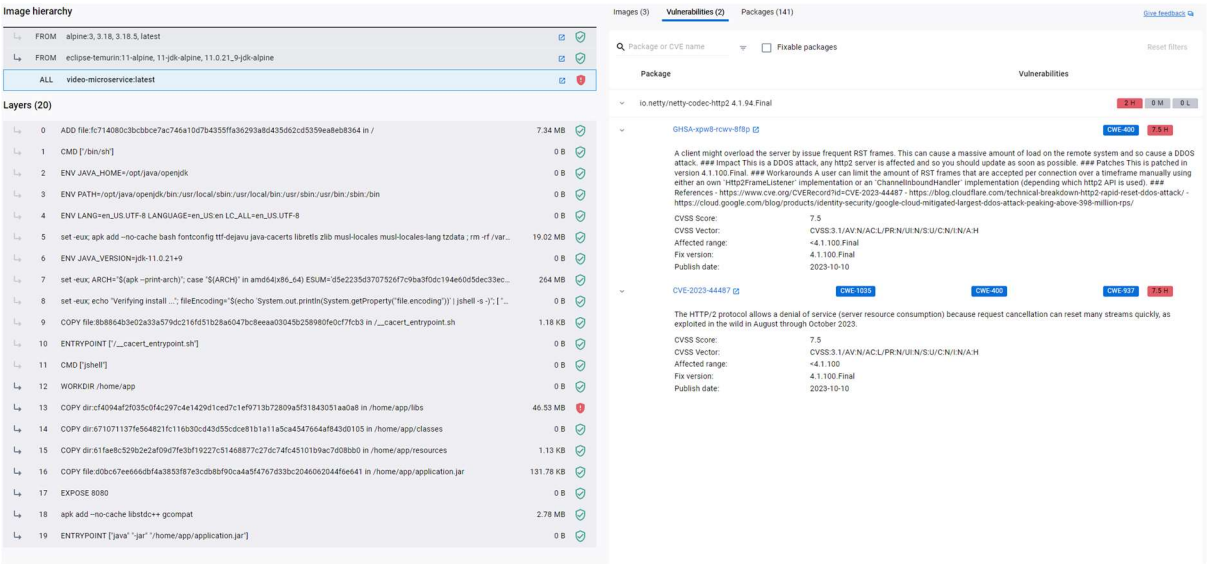
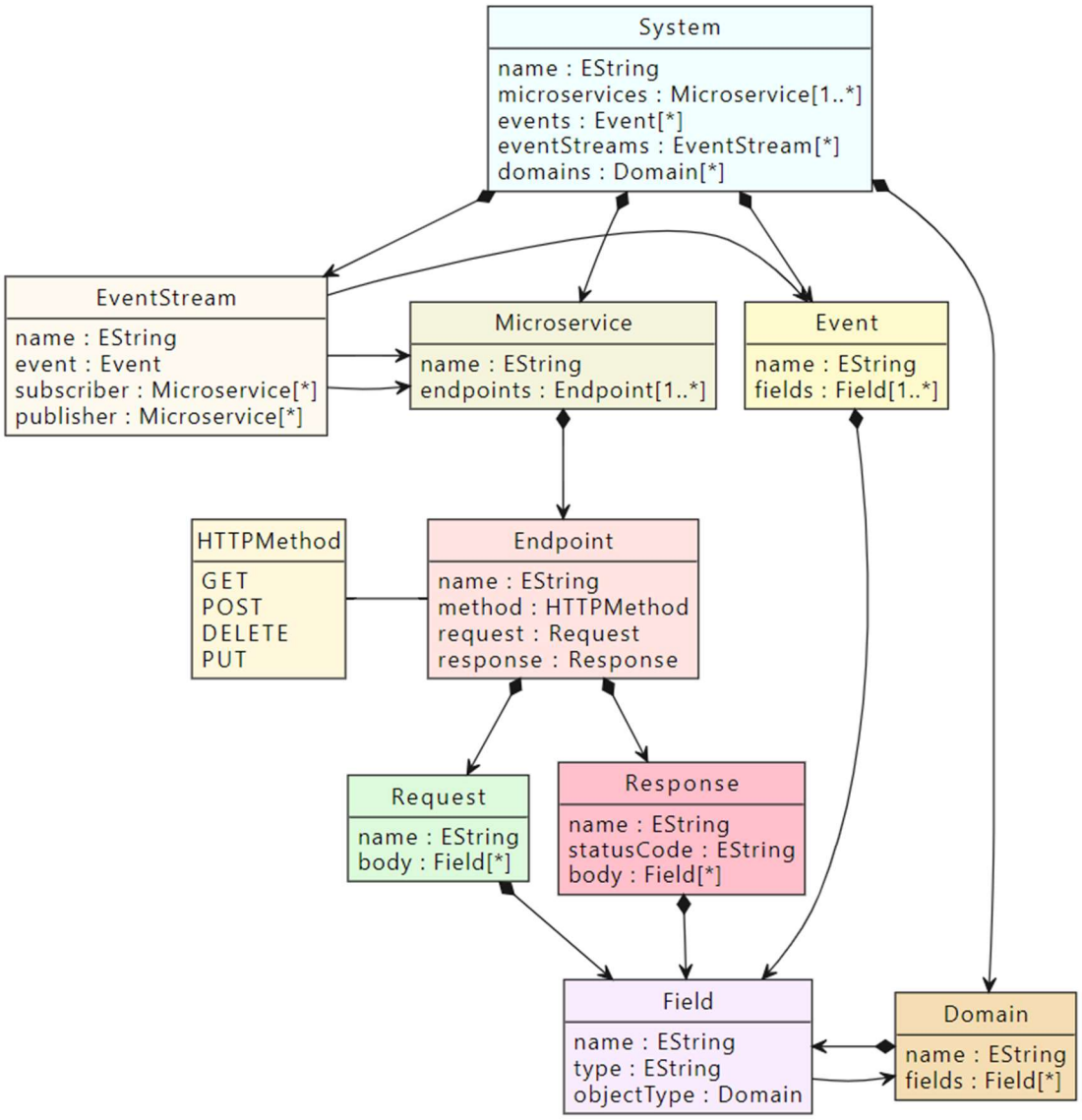


Figure 2: video-microservice image vulnerabilities post fix

Both the trending microservices and the subscription microservices has two-high level vulnerabilities. The Kafka image has a critical vulnerability but there was no fix version I could upgrade to this was left.

2.2 Application of Model-driven Engineering

2.2.1 Metamodel



My model consists of a top level `System` that defines the microservices, events and domain. The domain represents objects stored in a database such as a Video, Hashtag and User.

The system must have at least one microservice, which is defined by a name and a series of endpoints. An endpoint contains a name, an Enum HTTP method and a request and response. The request and response are almost identical, both taking a set of fields with the response taking an additional HTTP status code. A field consists of a name, datatype and an object type. This addition of the object type allows us to represent a domain as a field, e.g. a `list-video` endpoint could have a response body = {name: `videos`, type: `Set`, objectType:`Video`}.

The system also defines a set of events and event streams. An event is simply a set of fields and a name. This event is associated with at least one event stream: which consists of a name, an event, and references to the microservices it publishes the event from and subscribes to.

I decided to not include the Command Line Interface (CLI) in this model representation as the emf makes it clear that the microservices communicate using events and the endpoints are clearly stated within the microservices, so did not feel the need to include this.

2.2.2 Graphical Concrete Syntax

I used different colours: red and purple to represent the microservices and the event streams respectively so that it would be easy to distinguish them and create a higher visual distance. However, the use of the gradient from white to the colour does make it harder to distinguish than a solid colour. The fields were represented as a solid block colour to make them stand out as they are the smallest symbol. They are also in the same colour as their parent objects (blue request, blue response, green events) to make it clear that they belong to that container. It would have been better to use a different shape to represent the fields, to increase the visual distance since all my shapes are the same.

In the endpoints' containers, a request and a response are both represented by a white to blue gradient rectangle. They are differentiated by the direction of the gradient and their vertical position: the request box is always above the response rectangle. A weakness of this decision is that because they are both white to blue gradient boxes it may be difficult to separate them.

I represented the system using containers, to remove complexity within the diagram. Make an endpoint a subset within the microservices make it clear that those endpoints belong to a specific microservice, and a set of fields belong to a specific endpoint.

The relationship between the microservices and event streams are represented by arrows coming out of the two containers. A red arrow going out of a microservice and to an event stream represents which microservice the event streams are being published from and a grey arrow going out of an event stream to a microservice represents which microservice is subscribing to that event stream. A weakness of this is that it is difficult to know one first glance what the arrow represents if the viewer is not familiar with the model, it would have been beneficial to include 'subscribes to' and 'published from' labels to the arrow.

2.2.3 Model Validation

One constraint I made was to check that all events defined in the model are used in at least one stream. This was to make sure that there would not be any redundant events defined in the model. For this constraint, in the pre-condition I set a variable with all the event streams called streamedEvents, which was called in the event check. I checked that each event was included in streamedEvents.event.

On a system level, I checked that there was at least one microservice. This was done by getting all the microservices and checking the size was greater than zero.

On the event stream level, I had two constraints. I checked that for every event stream the number of microservices that subscribe to that event was at least one and that the number of microservices that publishes that event was at least one.

Finally, on a microservice level I checked that each microservice had a health endpoint. This was accomplished by collecting all the endpoints in the microservice where the HTTP method was get and the size of the requests body was zero. I then checked that the number of collected endpoints was greater than zero.

2.2.4 Model-to-Text Transformation

I chose to generate the docker compose file. The EGX file takes a system and transforms each microservice into a docker image, it also takes in a defined number of Kafka clusters and generates the images for each Kafka node. A downside to passing the number of Kafka nodes in the EGX file means that if we want to increase the number of nodes, we will have to manually change this in the EGX file. However, generating the docker compose file using the model to text transformations means that the docker file can be easily configured as we scale the system and increase the number of microservices, the docker file and images can be configured automatically.

The generated code is organised into two files. The compose file and the database generation sql file. The compose.yml file defines the database image, calling the generate SQL file to generate the different databases for each microservice, followed by the images for each microservice, before generating each cluster for the kafka-instances.

The generated sql lives inside a new folder, init, within the src-gen folder. The transformation takes each microservice and creates a database with that microservice name and grants all permissions for the database to the database user.