DEPARTMENT OF COMPUTER SCIENCE

Engineering 2: Automated Software Engineering (ENG2)

Exam No. Y3886186

1.0.0 | Build Guide

video-microservice (cd in the dir first)

build: ./gradlew build

docker image: ./gradlew jibDockerBuild

trending-microservice (cd in the dir first)

build: ./gradlew build

docker image: ./gradlew jibDockerBuild

subscription-microservice (cd in the dir first)

build: ./gradlew build

docker image: ./gradlew jibDockerBuild

client (cd in the dir first)

build: ./gradlew build

docker (cd in the dir first)

orchestration: docker-compose up

2.1.1 | Architecture

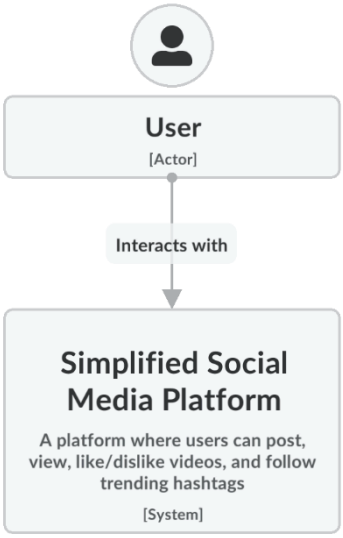


Figure 1 Context diagram

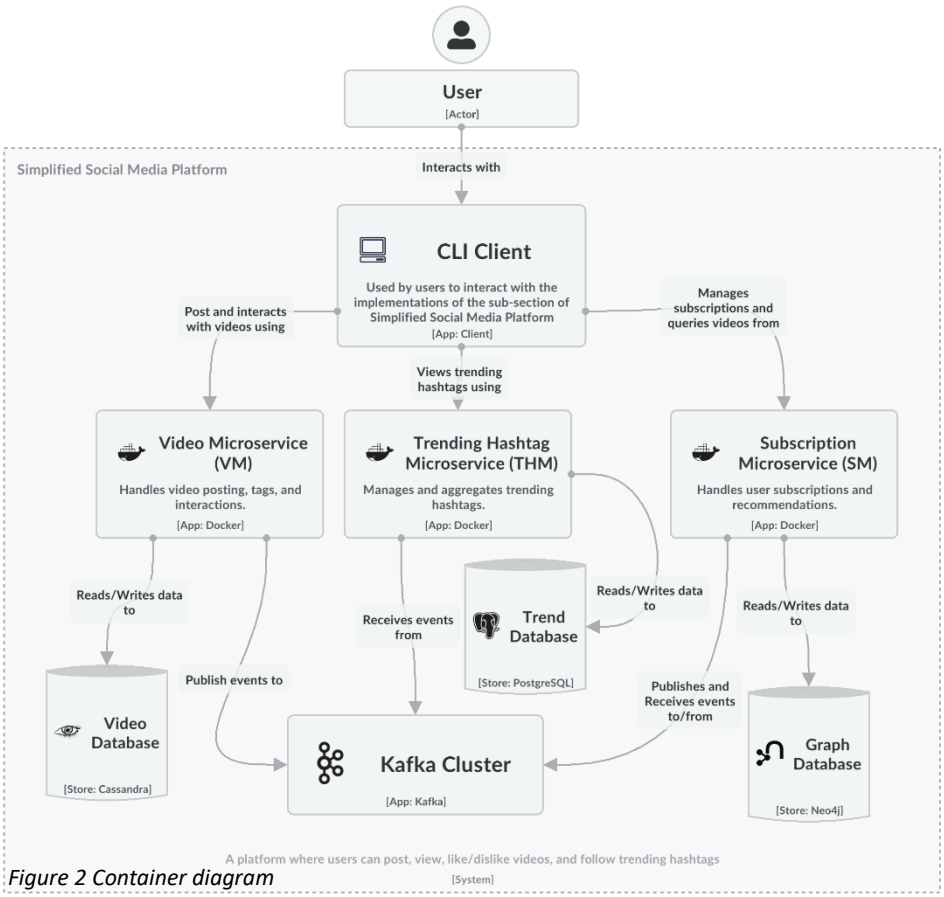


Figure 2 Container diagram

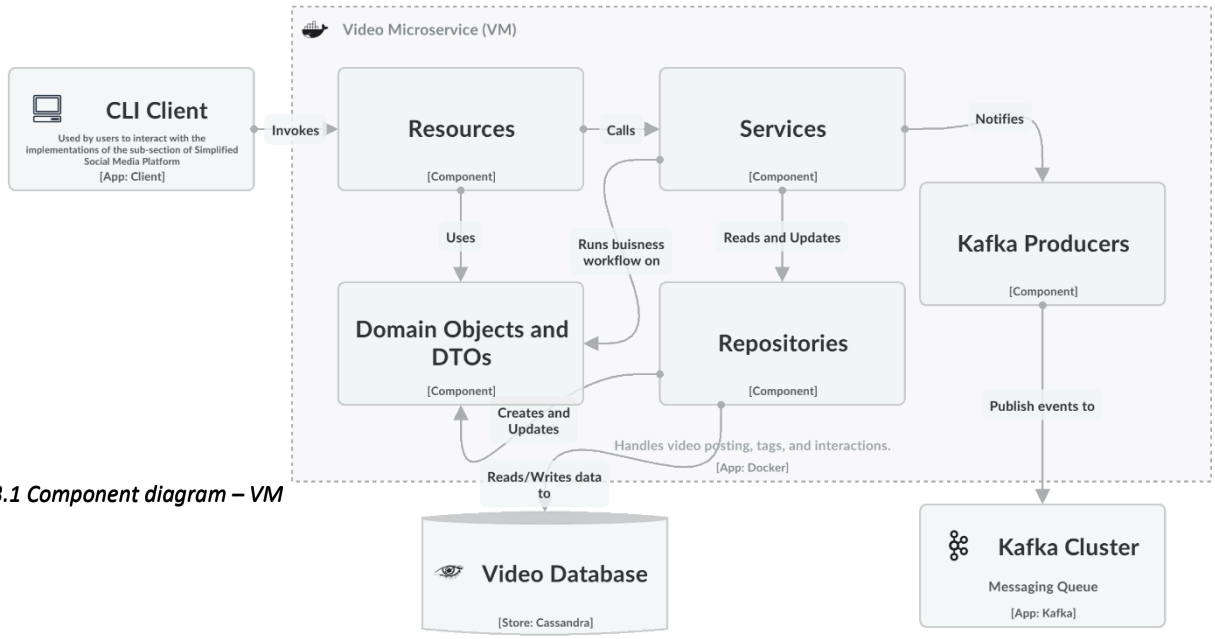


Figure 3.1 Component diagram – VM

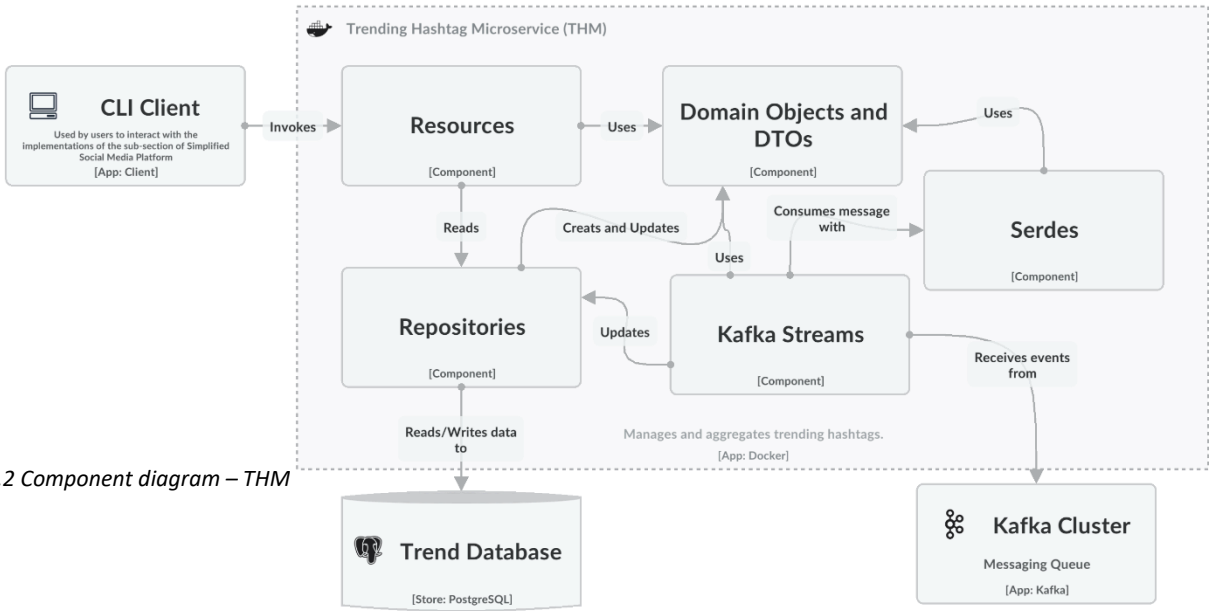


Figure 3.2 Component diagram – THM

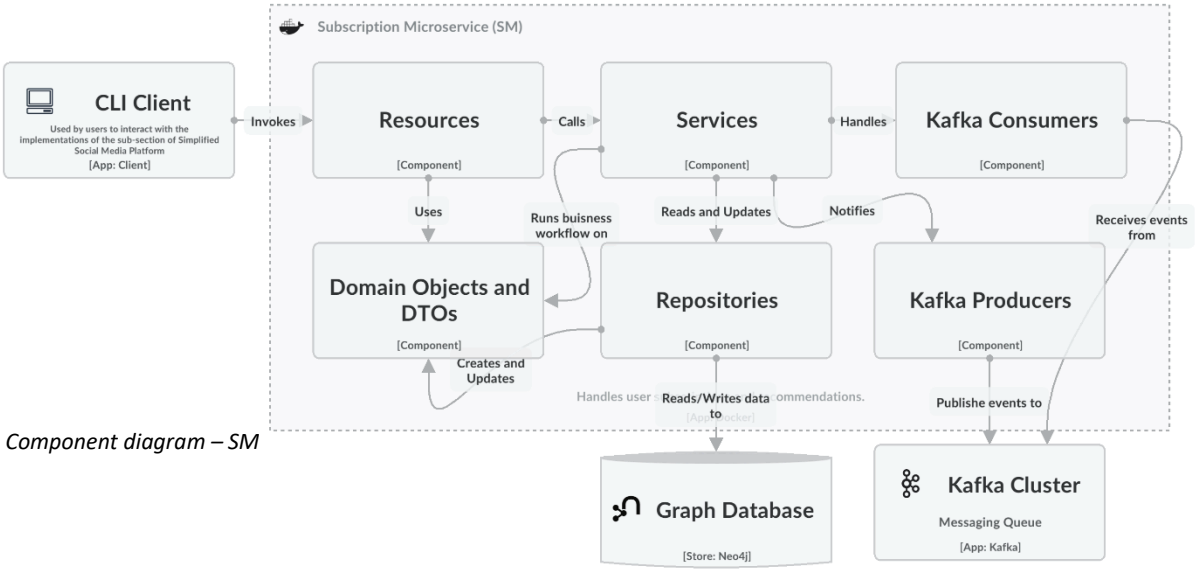


Figure 3.3 Component diagram – SM

The C4 diagrams provided above include context, container, and different component diagrams, collectively representing a software architecture designed to serve as a modern, data-intensive platform for social media. This architecture is tailored for high performance, scalability, and real-time data processing, leveraging a combination of advanced technologies and architectural patterns.

The primary objective of this architecture is to be inherently scalable and adaptive, equipping it to handle increasing user demands and future requirements, such as integrating a recommendation system. Key characteristics contributing to its scalability and adaptability include:

- Microservices Architecture: Microservices, which are self-contained, modular, and inherently scalable. This means that if demand for a specific service (e.g., the Video Microservice) increases, it can be scaled horizontally (by adding more service instances) without impacting other services. This approach is crucial for managing peak loads and accommodating a growing user base.
- Distributed Data Stores: Cassandra, used in the Video Microservice for its scalability and fault tolerance, effectively handles large data volumes across multiple servers with high availability. PostgreSQL, utilized for storing top liked hashtags over an hour, requires less scalability but can be seamlessly replaced with TimescaleDB for complex analytics and greater scalability. Neo4j is a graph database adept at managing complex, interconnected data, and scales well both vertically and horizontally with sharding and federation capabilities, providing unlimited scale-out.
- Kafka for Event Streaming: Kafka, a distributed event streaming platform, is integral for high data throughput management in real-time data processing applications. It scales effectively to accommodate growing user interactions and data streams.
- Modular Design: The microservices architecture facilitates the addition of new services and modification of existing ones without extensive redesign. For instance, integrating a recommendation system can be achieved by adding a new service that leverages existing data streams and user interactions.
- Technology Agnostic Interfaces: The use of RESTful APIs with Kafka as communication channels enables technology-agnostic interfaces among microservices. This design allows for the introduction of new technologies and the replacement of existing ones with minimal service logic modification. For example, Cassandra or Neo4j can be swapped for a more efficient database system with relative ease.
- Flexible Data Infrastructure: The existing infrastructure, particularly the use of Kafka and databases like Neo4j, lays a solid foundation for a recommendation system. Neo4j's graph capabilities are ideal for discerning user preferences and relationships, while Kafka can stream user interaction data in real-time to the recommendation engine, facilitating the integration of AI/ML algorithms for advanced personalized recommendations.

2.1.2 | Microservices

The simplified social media platform involves three main services: Video Microservice (VM), Trending Hashtag Microservice (THM), and Subscription Microservice (SM). Each microservice has its distinct responsibilities and technology stack, contributing to the overall functionality of the platform. Here is a high-level overview of each microservice:

- Video Microservice (VM)
    - Purpose: Handles all operations related to videos, including posting, listing, watching, and engagement (likes/dislikes).
    - Technology Stack:
        - Micronaut: A modern, JVM-based framework for building modular, easily testable microservices.
        - Cassandra: A distributed NoSQL database, used for storing videos, tags, and watch history.
        - Kafka: A distributed event streaming platform used as a messaging queue.
    - Responsibilities:
        - Manage video data and metadata.
        - Track user engagements and feedback on videos.
        - Publish events related to video interactions.
    - Components: Includes controllers for video operations, DTOs for event communication, repositories for Cassandra data management, producers for event publishing, and services for business logic implementation.
- Trending Hashtag Microservice (THM)
    - Purpose: Identifies and provides the current top 10 liked hashtags within a specified time window.
    - Technology Stack:
        - Micronaut: As the framework for building the microservice.
        - Kafka Streams: To process streams of data, particularly for aggregating top tags.
        - Postgres: A relational database to store aggregated data.
    - Responsibilities:
        - Aggregates likes per tag for rolling window of one minute.
        - Capability to perform analytics with custom time window (e.g. to get hourly top)
        - Subscribes to VM events to update trending hashtags dynamically.
    - Components: Includes controllers for managing hashtag-related requests, DTOs for event data transfer, a repository for Postgres database interactions, stream processors for real-time hashtag analysis, and utility classes for time interval conversion and validation.
- Subscription Microservice (SM)
    - Purpose: Manages user subscriptions to hashtags and recommends videos based on these subscriptions.
    - Technology Stack:
        - Micronaut: As the framework for building the microservice.
        - Neo4j: A graph database, ideal for managing relationships (like subscriptions).
        - Kafka: For publishing/subscribing event to messaging queue.
    - Responsibilities:
        - Handles user subscriptions/unsubscriptions to hashtags.
        - Uses VM events to update the relations in the graph.
        - Maintains a list of top videos to recommend based on subscriptions and/or user interaction with different videos.

- Components: Includes controllers for handling subscriptions and recommendations, DTOs for data exchange, mappers for converting between entities and DTOs, Neo4j repositories for data storage, producers for event publishing, consumers for event subscription and services for implementing business logic.

Inter-service Communication and Event Flow

Kafka Events: VM publishes events (video posted, liked/disliked, watched) that THM and SM subscribe to. SM publishes events (subscribed/unsubscribed) that has got no subscriber as of now.

Data Flow:

- VM updates its data in Cassandra and publishes relevant events.
- THM listens to liked/disliked event, processes the data using Kafka Streams, and updates Postgres with per minute aggregation.
- SM listens to VM events, updates the nodes and relationships in Neo4j for enhanced recommendation.

Command-line Interface (CLI) Client

Purpose: To allow a medium for an end-user to interact with the Simplified Social Media Platform. It acts as a middleman that can forward the user's arbitrary request to the designated microservice.

The Client has got the different endpoints for the microservices, and communicate with them over a request-response RESful APIs. The endpoints in future can be easily swapped by load-balancer for distributing the load evenly among different instances. It has got all the functionality implemented as individual commands, where the user can provide the command name and the contextual information associated to the command and then the CLI can display the response in a meaningful manner after contacting the designated microservice associated to the command. Available commands can be found by running CLI Client and passing `--help` option. Likewise, available flags and description for each command can be found by passing its name alongside `--help` option.

Usage:

- Post a Video: cli post [-hV] [--verbose] -t=<title> -u=<userId> -T=<tags> [-T=<tags>]...
- Like a Video: cli like-video [-hV] [--verbose] [-u=<userId>] [-v=<videoId>]
- Dislike a Video: cli dislike-video [-hV] [--verbose] [-u=<userId>] [-v=<videoId>]
- Watch a Video: cli watch-video [-hV] [--verbose] -u=<userId> -v=<videoId>
- List Videos posted by a user: cli user-profile [-hV] [--verbose] -u=<userId>
- Show Trending Hashtags: cli current-top [-hV] [--verbose] [-l=<limit>]
- Show Past Trends: cli past-top [-hV] [--verbose] [-i=<interval>] [-l=<limit>]
- Subscribe to Hashtag: cli subscribe [-hV] [--verbose] -t=<tagName> -u=<userId>
- Unsubscribe to Hashtag: cli unsubscribe [-hV] [--verbose] -t=<tagName> -u=<userId>
- List Recommended Videos: cli suggest-videos [-hV] [--verbose] [-t=<tagName>] -u=<userId>
- Timeline: cli timeline [-hV] [--verbose] -u=<userId>

The CLI can be built for different native platform for better integration with different operating system and providing better connectivity for the application with the help of Micronaut framework.

2.1.3 | Containerisation

Docker is an essential tool in modern software deployment since it encapsulates applications and their dependencies in containers, maintaining consistency across several environments. On the other hand, Docker Compose makes it easier to orchestrate these containers and is used to define and operate multi-container Docker applications. Every microservice, as well as its dependencies (such as Kafka, Cassandra, and so on), is containerized. By running a simple command (docker-compose up), all services can be launched concurrently while retaining their interdependence.

This strategy is extremely useful for development, testing, and production environments since it ensures that the software operates the same at each step. Furthermore, utilizing Google Jib to create efficient and optimized Docker images for Java applications improves the deployment process as it provides fast, reproducible and daemonless build features.

The Docker Compose solution can scale up to larger numbers of users, and be resilient to failures.

Scalability: It is a critical aspect of any modern application, especially one expected to handle a large and growing user base. Docker and Docker Compose support scalability in several ways:

- Microservice Architecture: Different application components can scale independently of one another in response to demand due to the nature of microservices (VM, THM, SM). For example, if the video service has a high volume of traffic, it can be scale separately without touching any other services.
- Service Replication: Docker Compose allows for straightforward service replication. For example, more instances of Kafka or Cassandra or Neo4j can be deployed to manage rising load.
- Load Balancing: The system's capacity to manage heavy traffic can be improved by integrating Docker with load balancers, which effectively distributes requests over several containers.

Resilience to Failures: It is crucial for maintaining the availability and reliability of the system. Docker and the microservices architecture contribute to this in several ways:

- Container Isolation: The entire system is not brought to a halt when a service within a container fails or crashes and without affecting other services, the compromised container can be swiftly restarted or replaced.
- Health Checks and Restart Policies: Configuring health checks and restart policies for containers is possible using Docker Compose. If a container becomes unhealthy or fails, it can be restarted automatically based on the configured policies.
- Distributed Systems and Replication: Using distributed databases such as Cassandra and Kafka ensure that the system is not vulnerable to a single point of failure. Moreover, data replication across nodes offers resilience to hardware problems.
- Decoupled Architecture: By separating services, the microservices architecture promotes resilience by design. Failures in one microservice (such as THM) have no direct influence on others (such as VM).

In the future, incorporating Kubernetes or a comparable orchestration tool may provide additional management tools for large-scale deployments. Kubernetes excels at orchestrating containerized applications across a cluster of servers, including features such as automatic rollouts, self-healing, and advanced scaling. It guarantees high availability and optimal resource use, making it a forward-thinking solution for changing application requirements.

video-microservice

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed
com.example		90%		100%	1	6	2	19	1	5	0
com.example.repository		100%		100%	0	32	0	176	0	26	0
com.example.service		100%		100%	0	29	0	105	0	22	0
com.example.controller		100%		100%	0	32	0	87	0	21	0
com.example.dto		100%		n/a	0	60	0	44	0	60	0
com.example.mapper		100%		n/a	0	9	0	33	0	9	0
com.example.model		100%		n/a	0	29	0	21	0	29	0
Total	5 of 1,905	99%	0 of 50	100%	1	197	2	485	1	172	0

trending-microservice

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed
com.example		37%		n/a	1	2	2	3	1	2	0
com.example.validator		97%		83%	5	20	1	24	0	4	0
com.example.stream		100%		91%	1	20	0	48	0	14	0
com.example.controller		100%		100%	0	11	0	23	0	6	0
com.example.dto		100%		n/a	0	14	0	13	0	14	0
com.example.util		100%		83%	1	6	0	13	0	2	0
com.example.model		100%		n/a	0	7	0	5	0	7	0
com.example.consumer		100%		n/a	0	2	0	3	0	2	0
com.example.repository		100%		n/a	0	1	0	1	0	1	0
com.example.serde		100%		n/a	0	1	0	2	0	1	0
Total	8 of 601	98%	7 of 58	87%	8	84	3	135	1	53	0

subscription-microservice

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed
com.example		37%		n/a	1	2	2	3	1	2	0
com.example.repository		100%		100%	0	64	0	160	0	58	0
com.example.dto		100%		n/a	0	62	0	47	0	62	0
com.example.service		100%		100%	0	28	0	72	0	24	0
com.example.controller		100%		100%	0	14	0	36	0	11	0
com.example.consumer		100%		100%	0	6	0	20	0	5	0
com.example.mapper		100%		n/a	0	9	0	27	0	9	0
com.example.model		100%		n/a	0	18	0	15	0	18	0
Total	5 of 1,899	99%	0 of 28	100%	1	203	2	380	1	189	0

The initial phase involved unit and integration testing, targeting individual components and their interactions. The primary objective was to ensure functionality, robustness, and reliability at the code level. JaCoCo, a code coverage tool, was instrumental in this phase, providing insights into the effectiveness of these tests. This report was included for each microservice in source code.

Test Results and Analysis

- Video Service (VM): High coverage with 99% of instructions and 100% of branches covered. This reflects that no branches within the code base was left unchecked and the code can be trusted for its functionality with high degree of confidence. The missed code coverage comes from application start-up test with custom arguments, but since this requires an independent run, it was interfering with Kafka configuration, resulting in list of “initializationError” errors. As the other integration tests covered this bit already, the corresponding test was commented out.

- Trending Hashtag Service (THM): Attained 98% instruction coverage and 87% branch coverage. The lower branch coverage was carefully invested for decision paths and complex logic, and it was found that all the other branches were unreachable and untestable unless the modifier of the classes were made more accessible, but it was ruled out unnecessary. The top 10 hashtag of current hour window (0^{th} min – 59^{th} min) test worked on the real-world clock, and unless the clock inside the database itself was mocked it would be flaky. Due to this reason, the "testGetTop10TagsOfCurrentHour" test only passed if minute hand showed 15 or more.
- Subscription Service (SM): Achieved a high level of code coverage with 99% of instructions and 100% of branches covered. Like VM; THM and SM also suffered from the "initializationError" errors on testing the application startup. Excluding that, the tests covered everything and ensured the code was reliable and robust with no missing edge cases for branching.

System Testing and Qualitative Assurances

Beyond code coverage, system testing was conducted using a script to interact with various microservice endpoints. This was done as an alternative to component testing, as that would have been resource extensive and trouble to set-up within the time frame. This phase focused on:

- Database Population and Integrity: All databases were correctly populated with accurate data, indicating successful data persistence and integrity.
- Data Transmission via Kafka: Seamless data flow between services, with no noticeable lag or loss, highlighting the efficiency of Kafka.
- Endpoint Functionality: All microservice endpoints responded correctly, indicating proper request handling.
- Top 10 Hashtags Functionality (THM): Accurate and timely updates of top hashtags, reflecting the dynamic user interactions.
- Recommendation System Accuracy (SM): The system showed high relevance and personalization in video suggestions, aligning well with user preferences.

Recommendations and Conclusion

The combination of code-level testing and system-level qualitative analysis gave a comprehensive picture of the microservices' functioning and robustness. While the test coverage figures were good, and the system testing exponentially increased the reliability, yet it is significance to improve on the current test cases. The future directions include:

- Automating System Tests: To ensure consistency and efficiency in testing.
- Expanding Test Scenarios: To cover more complex interactions and edge cases.
- Conducting Performance and Load Testing: Essential for understanding the system's behaviour under stress.
- Ongoing Monitoring and Improvement: Regularly updating tests in response to new features and changes in the system.

To sum up, the extensive testing approach used gave rise to significant trust in the microservices architecture's functionality and dependability, laying the groundwork for a robust and user-focused social media network. The system's resilience will be increased even more by ongoing testing methodology advancement.

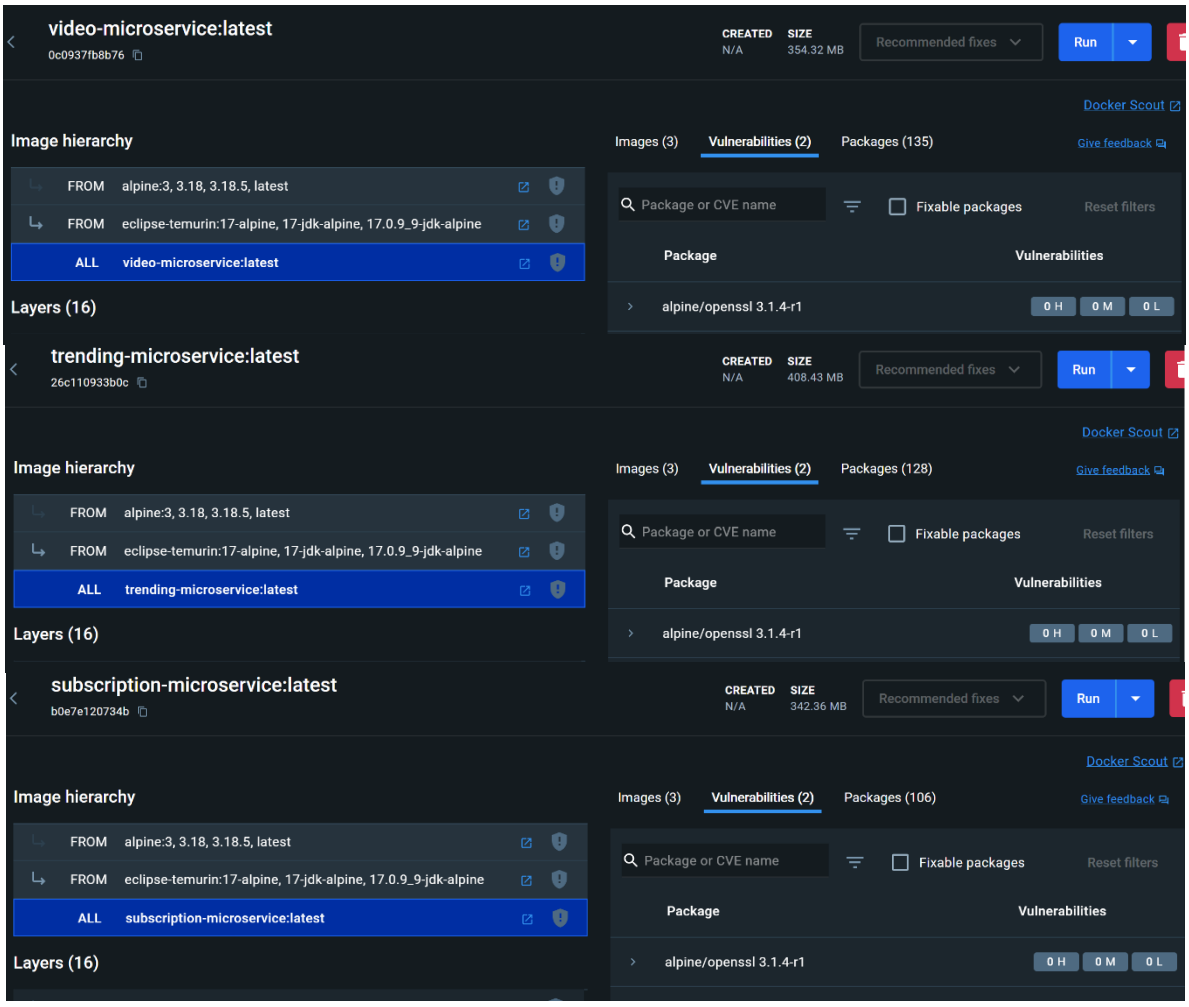
Initially, the Micronaut Docker plugin (./gradlew dockerBuild) was used to build the images. Trivy, an open-source comprehensive vulnerability, sensitive information, and secrets scanner, identified a total of 106 vulnerabilities (UNKNOWN: 0, LOW: 41, MEDIUM: 65, HIGH: 0, CRITICAL: 0). These vulnerabilities upon investigation were found to be embedded in the image, 'eclipse-temurin:17-jre-focal' used by the Micronaut Docker plugin.

To address these vulnerabilities, a switch was made to Google Jib (./gradlew jibDockerBuild), specifying 'eclipse-temurin:17-jdk-alpine' as the base image. This change significantly reduced the number of vulnerabilities to a total of 4 (UNKNOWN: 0, LOW: 0, MEDIUM: 4, HIGH: 0, CRITICAL: 0), primarily affecting the OpenSSL library (libcrypto3 and libssl3) in the Alpine 3.18.5 environment. The severity of these vulnerabilities was medium.

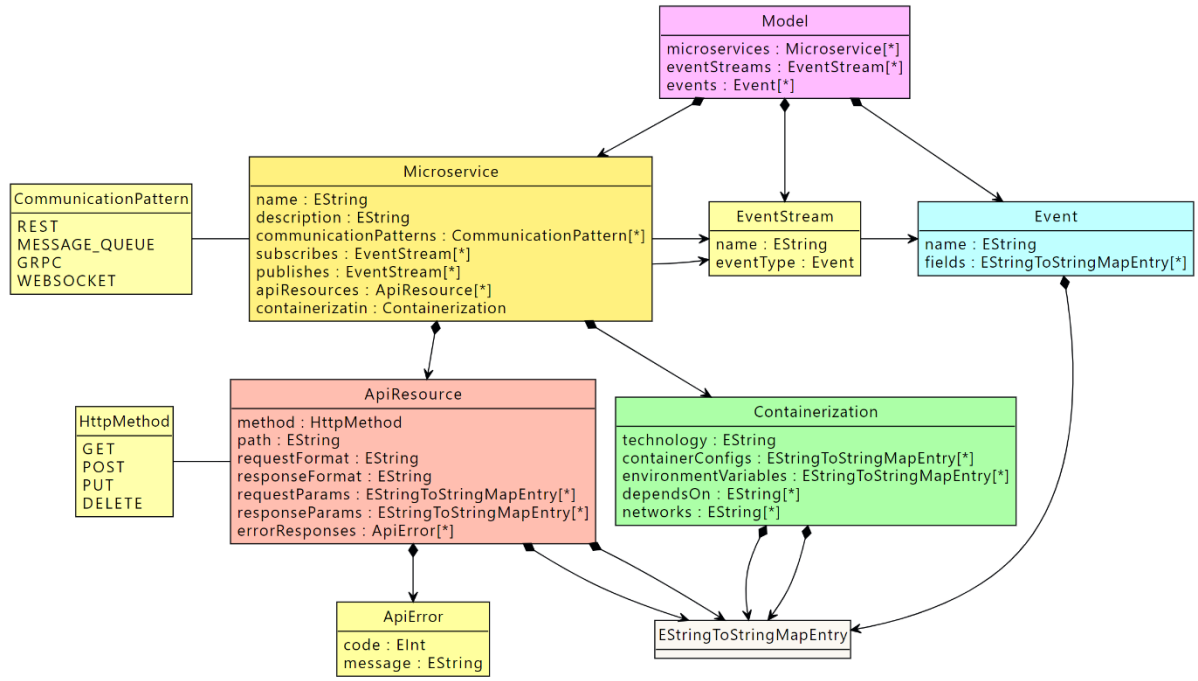
On scanning vulnerabilities within Jar files, a critical issue was identified in logback-core-1.4.13.jar (CVE-2023-6481) with high severity. This issue was present in all the microservices. To mitigate this, the version of logback-classic was upgraded to 1.4.14, which had addressed this vulnerability.

The Docker images for all three microservices (VM, THM, SM) underwent a similar process, ensuring a consistent security posture across the board. The OpenSSL vulnerabilities (CVE-2023-6129 and CVE-2023-6237) remained but were fixed in later versions. As these were dependencies of the Alpine 3.18.5 environment and were integral to it, a decision was made, based on a risk assessment considering the project's scope and time constraints, to not update them.

These results were then cross-validated using Docker Scout for assurance and its screenshots are provided below, capturing zero vulnerabilities associated to any microservice (except the OpenSSL).



2.2.1 | Metamodel



The Emfatic metamodel for the domain-specific language (DSL) is intended to describe a microservices architecture centred on events, event streams, and microservices. This metamodel is designed to represent a data-intensive system's high-level structure, including events, their fields and types, event streams, microservices, API resources, and containerization information.

Discussion of the Metamodel

1. Events: This class is the foundation of the metamodel, reflecting the event-driven nature of current systems. Each event is identified by a unique name and a map of fields and their corresponding types, allowing for a wide variety of event structures. This flexible design is critical for accommodating many sorts of data that microservices could handle, from simple event to sophisticated data payloads.
2. Event Streams: Event streams are directly connected to certain event kinds, giving a clear link between the data flow and its structure. This relationship is critical for coordinating the flow of information across the system and ensuring that each microservice responds to the correct events. The explicit naming and linking of event streams to events aid in maintaining clarity and coherence in the architecture.
3. Microservices: This class contains a thorough representation of each individual service in the architecture. The inclusion of data such as name, description, and communication patterns provide a comprehensive view of each microservice. The relationships to event streams for subscription and publication demonstrate the interactive nature of microservices. In addition, providing API resources with characteristics such as HTTP methods, request/response formats, and parameters, as well as possible error replies, provides a complete blueprint of the microservice's capabilities and interfaces.

4. API Resources: This class is essential for defining the service interfaces. How external entities interact with the microservices can be precisely and clearly specified by providing information about the HTTP methods, paths, and parameters for requests and responses. This level of detail is required for API documentation, client integration, and establishing a contract between different components of the system.
5. Containerization: Including a class for containerization reflects an understanding of the importance of deployment and operational considerations in microservices architecture. Defining container technologies, configurations, environment variables, dependencies, and networks encapsulates the operational environment of each microservice, which is crucial for deployment, scaling, and management in a cloud-native context.
6. Communication Patterns: The enumeration of communication patterns like REST, MESSAGE_QUEUE, GRPC, etc. provides insights into how microservices interact with each other and external entities. This aspect of the metamodel is fundamental in designing the system's overall communication strategy, affecting latency, throughput, and scalability.

Assumptions Made

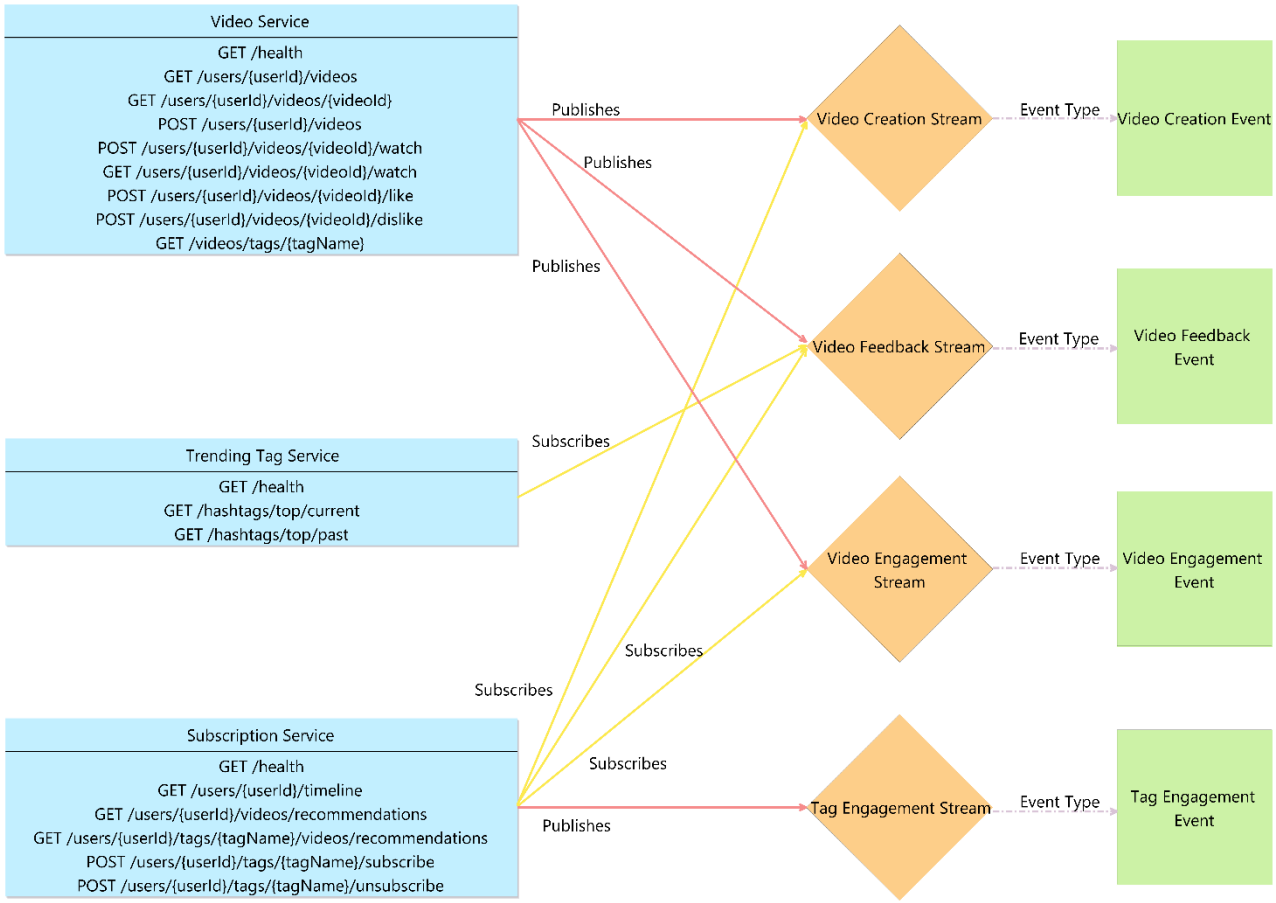
1. Simplicity: The metamodel assumes a certain level of simplicity and abstraction. For instance, it does not cover every possible HTTP method but focuses on the most used ones.
2. Flexibility and Scalability: The usage of maps for fields and parameters anticipates that events and API resources will vary widely and require flexibility to evolve. This approach assumes that events and APIs in a microservices architecture would evolve over time, requiring a flexible framework capable of accommodating additions or alterations without requiring large metamodel redesigns.
3. Modern Microservice Practices: The inclusion of containerization and various communication patterns assumes a modern approach to microservices architecture, potentially overlooking legacy systems or alternative architectures.

Alternative Design Decisions

The initial metamodel focused more on a generalized architecture representation, with entities, data flows, and a more abstract approach to microservices and events. However, this approach lacked specificity in terms of defining the structure and behaviour of events, event streams, and microservices, which are crucial for a DSL aimed at data-intensive systems. The current metamodel addresses these limitations by providing more detailed and specific structures for these components. The current metamodel is an iterative one, as this earlier only consisted of Event, Event Streams and Microservice, but on working with this, the need for other classes was noticed and thus more classes were appended later to the metamodel to fill the gaps. The same feedback loop can continue in the future to incorporate future requirements.

To summarize, the Emfatic metamodel provides a formal and thorough description of a microservices architecture in the context of data-intensive systems. It strikes a mix between flexibility, specificity, and contemporary microservice practices, making it an excellent choice for expressing such systems. The transition from a more abstract initial model to the current one demonstrates an evolution towards a more practical and applicable approach in the realm of domain-specific modelling for microservices.

2.2.2 | Graphical Concrete Syntax



Graphical Syntax Design and Implementation:

1. Semiotic Clarity: Distinct graphical elements are used to represent different elements of the model (e.g., Microservices, Events, Event Streams). Each symbol has a one-to-one correspondence with its referent concept, ensuring clarity in representation.
2. Perceptual Discriminability: Different colours and shapes are assigned to different elements (e.g., light green for events, light orange for event streams, and light blue for microservices containers). This enhances the visual distance between symbols, aiding in faster and more accurate recognition.
3. Graphic Economy: The number of graphical symbols is kept manageable. The use of simple shapes and limited colours adheres to the human span of absolute judgment, preventing cognitive overload.
4. Semantic Transparency: The use of intuitive shapes and labels (like using a square for API Resources with a label indicating method and path) enhances the intuitiveness of symbols. While some elements are semantically immediate, others might require domain knowledge for full comprehension.
5. Complexity Management: The design includes mechanisms like filters (e.g., Hide No Publishers, Hide No Subscribers) and toggleable Transition Layer to manage complexity and reduce cognitive overload. This aligns with principles like hierarchy, modularisation, and viewpoints.
6. Dual Coding: Text is used alongside graphical representations. For instance, labels on edges (e.g., “Publishes,” “Subscribes,” “Event Type”) complement the graphical information, providing a

more effective understanding than using graphics or text alone. Different colour (e.g., light red for “Publishes” and dark yellow for “Subscribes”) and pattern (e.g., dashes for “Event Type”) are also being used.

7. Toolkit: Different creation tools (e.g., Create Microservice, Create Event, Create Event Stream) has been provided so anyone with domain-knowledge could contribute to the model using visual alone.

Justification of Design and Implementation Decisions:

- The design choices made in the Sirius model are grounded in communication theory and the principles of effective graphical syntax design. For example, the decision to use different colors and shapes is rooted in perceptual discriminability.
- The use of filters and additional layers for managing complex models aligns with complexity management strategies. These features enable the model creator to focus on relevant parts of the model without being overwhelmed by its entirety.
- The encoding of the model allows for easy discovery, creation, and arrangement of symbols, adhering to the principles of encoding and decoding in diagram construction.

Strengths and Weaknesses:

Strengths:

- Clarity and Discriminability: The use of distinct shapes and colours aids in quick identification and differentiation of model elements.
- Complexity Management: Features like filters help in managing large and complex models, making them more manageable and understandable.
- Dual Coding: The combination of text and graphics enhances comprehension and retention.

Weaknesses:

- Learning Curve: New users might require time to familiarize themselves with the syntax and semantics of the graphical elements.
- Limited Flexibility: The graphical representation might be restrictive in expressing certain complex scenarios or abstract concepts compared to textual representations.
- Dependency on Visual Acuity: The effectiveness of the syntax heavily relies on the user's ability to perceive colour and shape differences, which might be a limitation for users with visual impairments.

Comparison with Alternatives:

Compared to textual syntaxes or other graphical modelling tools, the Eclipse Sirius-based syntax offers a more intuitive and visually engaging way to represent and understand complex models. However, it might be less flexible in representing abstract concepts and requires a certain level of visual acuity. Textual models, while less intuitive at first glance, can offer more detail and flexibility in certain contexts.

2.2.3 | Model Validation

Epsilon Validation Language (EVL) was used to implement validation constraints that cannot be directly expressed in the provided metamodel to ensure the integrity and correct functionality of the model elements.

Here is an analysis and implementation of each constraint:

UsedInAtLeastOneEventStream (Event Context):

- Rationale: This constraint ensures that each Event in the model is utilized in at least one EventStream. It is vital because an unused event could indicate an oversight or a redundancy in the model, making the model leaner and more purposeful.
- Implementation: The check clause examines all EventStream instances and ensures that the current Event (self) is linked as the eventType in at least one of them. If this condition is not met, an error message is generated, specifying the unused event.

HasAtLeastOnePublisher (EventStream Context):

- Rationale: This constraint assures that every EventStream has at least one Microservice publishing to it. This validation is critical for ensuring data flow in the system, as an event stream without a publisher would be redundant.
- Implementation: It checks through all Microservice instances to find at least one instance that includes the current EventStream (self) in it publishes collection. If no such microservice exists, an error message is generated, indicating the need for a publisher.

HasAtLeastOneSubscriber (EventStream Context):

- Rationale: This critique indicates that each EventStream should ideally have subscribers. This is not as critical as having a publisher, but it is still important for the effective use of event streams.
- Implementation: Like the previous constraint, but here it checks for inclusion in the subscribes collection of Microservice instances. If an event stream has no subscribers, a warning message is issued, suggesting the addition of at least one subscriber.
- Only recommendation message is emitted on failure of this constrain as there exists subscribed/unsubscribed event with no subscriber.

AtLeastOneMicroserviceExists (Microservice Context):

- Rationale: This constraint validates the existence of at least one Microservice in the model, which is fundamental for the operation of the system described by the model.
- Implementation: The check clause simply counts the total number of Microservice instances and ensures that this count is greater than zero. If no microservices are defined, an error message is generated.

HasHealthCheckResource (Microservice Context):

- Rationale: This constraint enforces that each Microservice should have a health check API resource. Such an endpoint is crucial for monitoring and maintaining the microservice's health and status.
- Implementation: It checks if there is at least one API resource within the Microservice that meets the following criteria: uses the HTTP GET method, has a path of '/health', and takes no request parameters. If such a resource is not found, an error message is produced.

2.2.4 | Model-to-Text Transformation

Model-to-text (M2T) conversions are important in software engineering, particularly when developing microservices. They make it easier to generate source code automatically from abstract models, which improves development efficiency and consistency. Epsilon Generation Language (EGL) is being leveraged to implement an M2T transformation, focusing on the domain-specific modelling language (DSML) provided, and aims to generate parts of the code for the Video Microservice (VM), Trending Hashtag Microservice (THM), and Subscription Microservice (SM) with the help of EGL Co-Ordination Language (EGX).

Model-to-Text Transformations in Detail

1. Microservice Scaffolding Generation:
    - Objective: To establish a foundational structure for each microservice, aligned with microservice architecture best practices. This includes generating source and test directories and basic ".keep" files for controllers, DTOs, services, and repositories.
    - Implementation Details:
        a. The EGX script reads each microservice definition from the DSML.
        b. For each microservice, it creates directories corresponding to the layers of the architecture (e.g., controller, model, service).
        c. Placeholder files are generated to maintain the directory structure in version control systems. This also helps in setting up a standardized project structure across all microservices.
2. API Controllers Generation:
    - Objective: To create Java controller classes that map to the API resources defined in the DSML. These controllers handle HTTP requests and responses.
    - Implementation Details:
        a. The EGX script processes the API resources section of each microservice.
        b. For each API endpoint, a corresponding method is generated in the controller class, annotated with the appropriate HTTP verb and path.
        c. These methods act as stubs, which can later be filled with the specific logic for request handling.
3. Event Handling Code Generation:
    - DTO Generation:
        a. Purpose: To create Data Transfer Object (DTO) classes that map to the event definitions in the DSML. These classes define the structure of data as it is transferred through the system, particularly in event streams
        b. Process: The EGX script generates a DTO class for each event type for every associated microservice, with attributes reflecting the event's data structure.
    - Kafka Listeners and Producers:
        a. Purpose: To enable asynchronous communication between microservices using Kafka.
        b. Process: For each event stream that a microservice subscribes to or publishes, the EGX script generates Kafka listener classes or Kafka producer interfaces. This facilitates the publishing and consumption of events to and from Kafka topics.
4. Health Controller and Test Generation:
    - Health Controller:
        a. Purpose: To generate Health Controller, offering an endpoint for checking the service's health status.

    b. Process: The EGX template generates a Health Controller for each microservice. This controller includes a predefined endpoint (/health) that returns the service's health status.
- Automated Generation of Test Cases:
    a. Purpose: Alongside the Health Controller, the system also generates test cases for these controllers.
    b. Benefits: Automated test generation saves significant time and effort that would otherwise be spent on manually writing tests. It ensures that basic testing is in place from the outset, contributing to a more robust and reliable microservice architecture.

5. Docker Compose Configuration:
    - Objective: To automate the generation of “docker-compose.yml” configurations for all microservice to orchestrate microservices deployments.
    - Implementation Details:
        a. The EGL template iterates over the microservices, extracting containerization details from the DSML.
        b. It generates a Docker Compose file with services configurations, including environment variables, port mappings, dependencies, and network settings.
    - Benefits: This ensures a consistent and error-free setup for containerization, crucial for deployment and scalability.

Justification of Generated Code Organization

1. Clarity and Consistency:
    - The code organization follows industry standards for microservices, ensuring that each component (controller, service, repository, etc) is clearly defined and separated. This makes the codebase more readable and maintainable.
2. Scalability and Flexibility:
    - The EGL templates are designed to be flexible and scalable. New services or modifications to existing services can be added to the DSML and reflected in the codebase through regeneration. This adaptability is crucial in a dynamic development environment.
3. Maintenance and Error Reduction:
    - Automated code generation minimizes human error and ensures consistency across various components of the microservices architecture. This reduces the likelihood of bugs and streamlines maintenance.
4. Customization and Extension:
    - Using EGL’s protected regions, developers can insert custom code that will not be overwritten in subsequent code generations. This feature allows for the extension of generated code to accommodate specific business logic or optimizations.

The usage of EGX and EGL for M2T transformations in this scenario demonstrates the power of model-driven engineering. By automating the process of creating a uniform, scalable, and maintainable codebase, developers can concentrate on complicated business logic and innovation rather than boilerplate code. The organized structure of the generated code, combined with the flexibility for customization, aligns well with modern software engineering practices, making it a robust solution for the development of a microservices architecture.