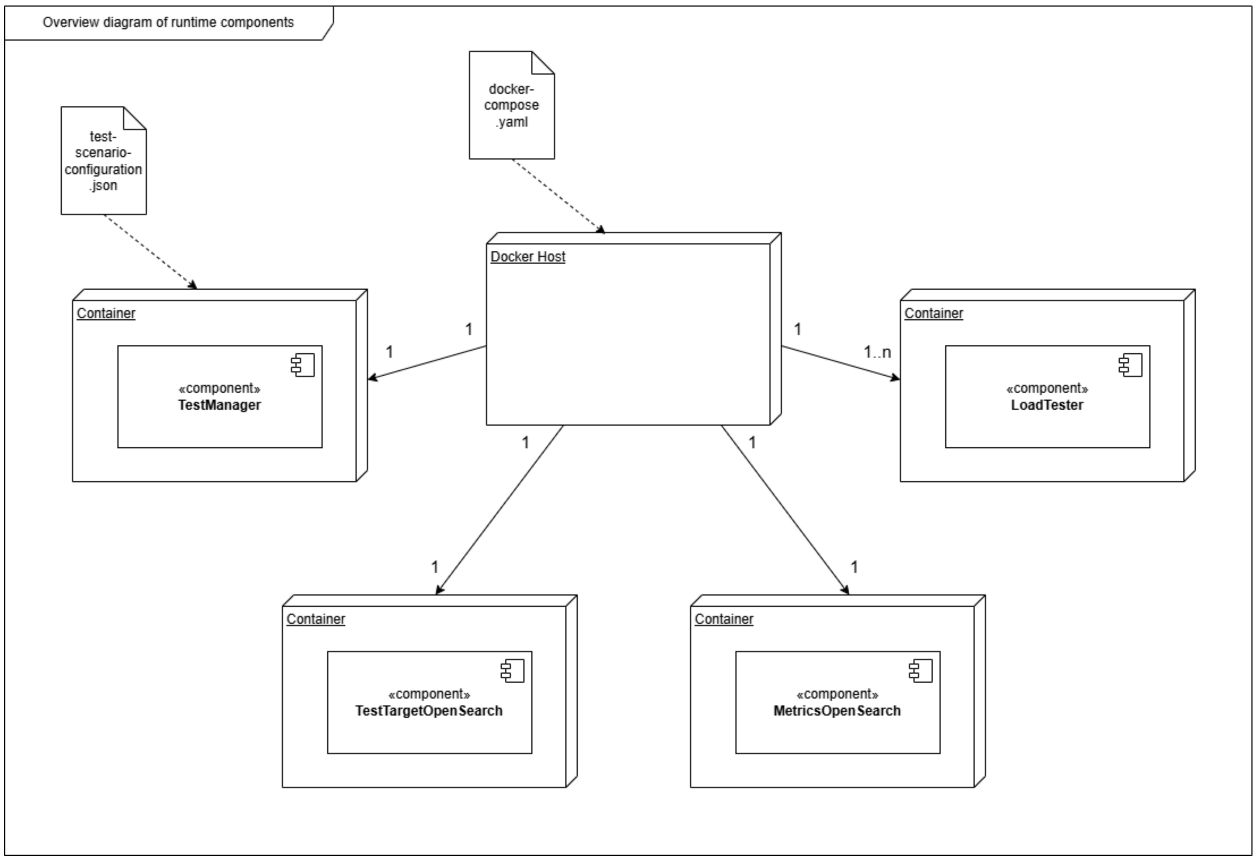
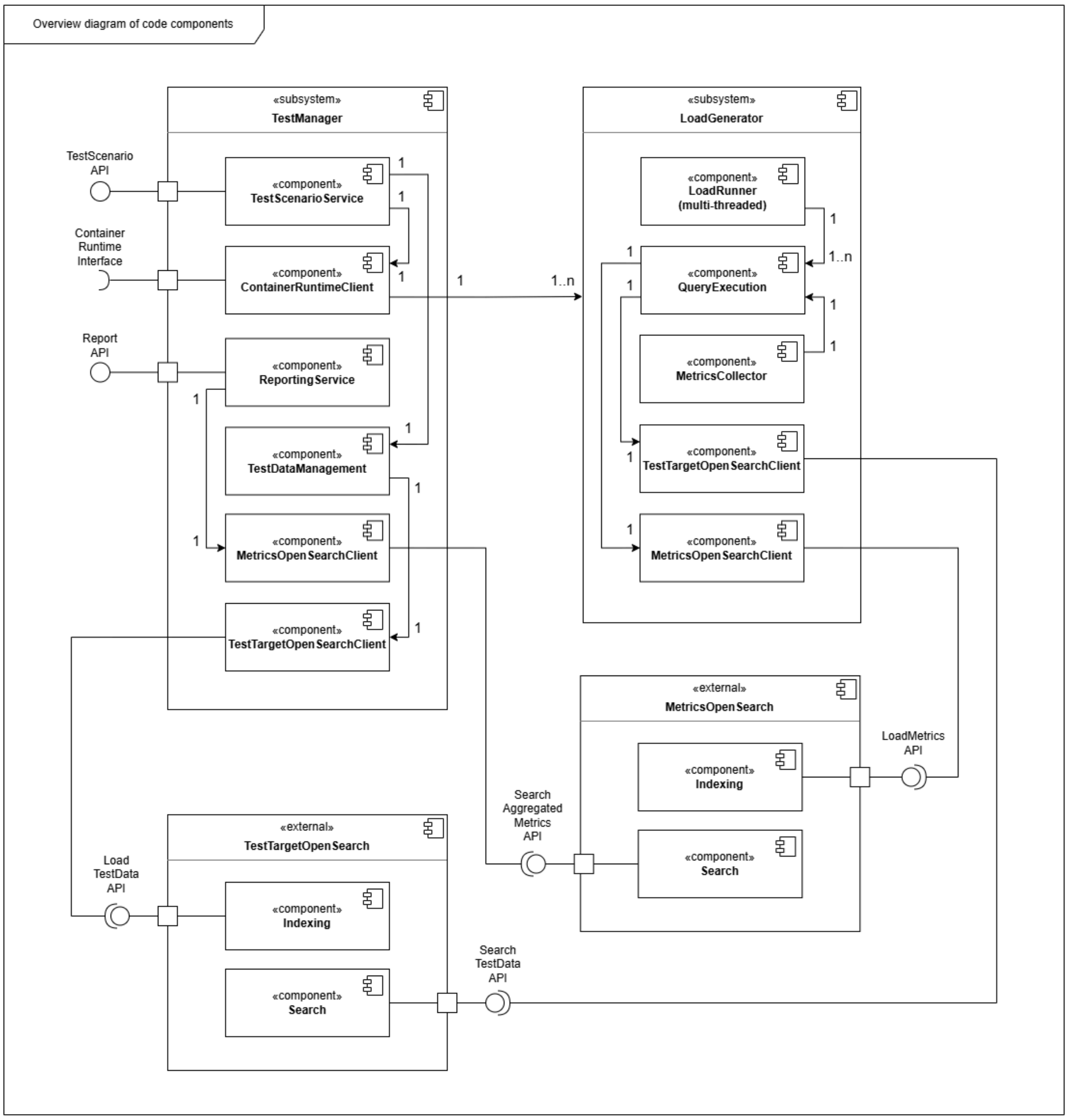
Overview diagram of runtime components


Overview diagram of runtime components
docker-compose.yaml
test-scenario-configuration.json
Docker Host
Container
«component» TestManager
Container
«component» LoadTester
Container
«component» TestTargetOpenSearch
Container
«component» MetricsOpenSearch
1
1
1
1..n
1
1
1
1

Overview diagram of code components


Overview diagram of code components
«subsystem»
TestManager
TestScenario API
Container Runtime Interface
Report API
«component» TestScenarioService
«component» ContainerRuntimeClient
«component» ReportingService
«component» TestDataManagement
«component» MetricsOpenSearchClient
«component» TestTargetOpenSearchClient
«subsystem»
LoadGenerator
«component» LoadRunner (multi-threaded)
«component» QueryExecution
«component» MetricsCollector
«component» TestTargetOpenSearchClient
«component» MetricsOpenSearchClient
1
1..n
«external»
MetricsOpenSearch
«component» Indexing
«component» Search
LoadMetrics API
Search Aggregated Metrics API
«external»
TestTargetOpenSearch
«component» Indexing
«component» Search
Load TestData API
Search TestData API

Summary of the underlying technology stack

The selection of technologies is based on the specifications and requirements of the IP. A primary premise was the use of Spring Boot. Additionally, it was mandatory that the application is containerized.

Java, Eclipse Temurin OpenJDK
Java v.25 was chosen as the programming language because it forms the technological foundation for the framework preferred by the IP. For the Java JDK, Eclipse Temurin 25 (LTS) was selected. This choice is justified by Temurin being an open source and production ready distribution of OpenJDK, which ensures stability and long term maintainability without license complications.

Spring Boot
The fundamental framework decision for both core components, the TestManager and the LoadTester, was Spring Boot. This choice was primarily made to ensure future maintainability and further development by the IP themselves. The IP's team already possesses extensive know-how with this technology, enabling independent project support after handover. Through core mechanisms like Dependency Injection and extensive auto configuration, complexity is significantly abstracted and the necessary configuration effort is reduced. This abstraction, combined with a variety of ready made modules for common tasks, significantly decreases the amount of boilerplate code.

Micrometer
Micrometer is employed for metric collection. The decision for Micrometer is due to its existing integration into the Spring Boot framework. This allows for standardized and consistent collection of application metrics with minimal configuration effort.

Apache Maven
Maven is used for dependency management and controlling the build process of the applications. This decision was based on its ability to standardize the build process and reliably manage all project libraries and their versions. This ensures reproducible builds and a consistent project structure, which is essential for maintenance.

OpenSearch
A separate OpenSearch instance serves as the data store for the collected test metrics. This architectural separation is necessary to reliably prevent performance interference with the actual test target, the OpenSearch Cluster, due to writing metrics. This externalization of data storage allows the LoadTester containers to send their data directly and operate statelessly. This is a fundamental prerequisite for achieving the required horizontal scalability of the LoadTesters.

Containerisierung: Docker
The TestManager, LoadTester and OpenSearch instances are each packaged in a Docker container. This containerization is the technical requirement for operation on the IP's Kubernetes cluster.

Textual explanation of the diagrams and choices

As shown in the *Runtime Components Diagram*, the applications run in containers on a container host, managed through configuration files. The *docker-compose.yaml* defines the container infrastructure, from which *TestManager*, *MetricsOpenSearch*, and *TestTargetOpenSearch* are derived. The *LoadGenerators* are created by the *TestManager* via the Docker Host based on the specifications in the *test-scenario-configuration.json*.

According to the *Code Components Diagram*, the *TestManager* is designed as a permanent management service responsible for orchestrating the entire test run. The *TestScenarioService* component manages the test configuration (e.g., #LoadGenerators, #queries, query type, etc.). Through the external REST interface *ScenarioAPI*, it receives the test configuration as a JSON file. Based on these specifications, it instructs the *ContainerRuntimeClient*, which encapsulates the communication with the Docker Host via the *ContainerRuntimeInterface*, to start, stop, or scale any number of *LoadGenerators*. The *test-scenario-configuration.json* should not only be passed through the REST interface but it can also be passed to the Docker Host, where it can be retrieved by the *ContainerRuntimeClient* through the *ContainerRuntimeInterface*. In both cases, the *TestScenarioService* is responsible for executing the test scenario. The *TestDataManagement* is instructed to generate, remove, or extend the test data for each test run. These data are then used to populate the *TestTargetOpenSearch* with data via the *LoadTestData* api through the TestTarget*OpenSearchClient*. The *ReportingService* is responsible for processing and providing the test results. Internally, it instructs the Metrics*OpenSearchClient* to query the aggregated metrics from the *MetricsOpenSearch* via the *SearchAggregatedMetrics* interface. A report can be retrieved as a JSON or CSV document through the *ReportAPI*.

The *LoadGenerator* contains the actual load generation logic. It is designed as a stateless component, enabling horizontal scalability. As illustrated in the *Code Components Diagram*, the *TestManager* can initiate any number of *LoadGenerators* to generate load on the *TestTargetOpenSearch*. The *LoadRunner* is responsible for executing queries in parallel to produce the defined load within a single instance. It coordinates the *QueryExecution*, which performs the requests on the *TestTargetOpenSearch* through the TestTarget*OpenSearchClient*. The results of these operations, such as response times and success status, are captured by the *MetricsCollector* and persisted in the *MetricsOpenSearch* via the Metrics*OpenSearchClient*.

The *TestTargetOpenSearch* represents the actual OpenSearch instance under test, while the *MetricsOpenSearch* instance serves as the storage location for the test results.