Link to GitHub Page:
https://github.com/amosproj/amos2025ws01-opensearch-load-tester/wiki/Design-Documentation

General Overview

The OpenSearch Load Tester consists of three core modules, each running as its own Spring Boot application:

- **Test Data Generator** - Responsible for creating and indexing synthetic test data in the target OpenSearch instance.
- **Load Generator** - Executes the actual load test by sending parallel queries to the target OpenSearch instance.
  It also collects performance metrics during execution and forwards them to the Metrics Reporter.
- **Metrics Reporter** - Collects and exposes all test metrics, making them easily accessible for analysis.

All three components, along with the OpenSearch test target, are orchestrated and managed using a single docker-compose setup.

Figure: OpenSearch Load Tester runtime architecture

Test Data Generator

The Test Data Generator consists of the following core code components:

TestDataInitializer

Implements a `CommandLineRunner` that executes the following steps at application startup:

1. Create the target index in OpenSearch.
2. Generate a provided number of test documents.
3. Bulk-index all generated documents into the target OpenSearch in a single request.
4. Refresh the index to ensure that the documents are immediately visible.

DataGenerator

Creates test documents with randomized values for the target OpenSearch instance.

- `DynamicDataGenerator`: Generates randomized documents without persisting them locally.
- `PersistentDataGenerator`: Generates and stores randomized documents locally (default output: /data/testdata.json).
  On subsequent executions, the generator reuses the existing file unless it is manually removed.

FileStorageService

Saves and loads test documents provided by the `PersistentDataGenerator`.

OpenSearchDataService

Provides operations for creating OpenSearch indices and performing bulk indexing of documents.

Load Generator

The Load Generator consists of the following core code components:

ScenarioConfigLoader

Reads test scenario configuration from a YAML file and maps it into a `ScenarioConfig` object.

ScenarioInitializer

Implements a `CommandLineRunner` that executes the provided test scenario at application startup.

LoadRunnerService

Handles the creation, execution and collection of execution threads. It is responsible for the Load Generator's workflow.
Implements an `execute()` method which executes a load test by given parameters.

QueryExecution Interface

Implements a `Runnable` `run()` method for one single execution of a query.
`OpenSearchQueryExecution` is a specialized class for OpenSearch queries.

MetricsCollector

Receives and stores metrics from each QueryExecution.

Metrics Reporter Client

When triggered, it sends all metrics data to the MetricsReporter component by HTTP request.

Metrics Reporter

Design overview of the Metrics Reporter service that collects load-test metrics, aggregates them, and exports JSON/CSV reports.

Purpose and scope

- Accept metrics from one or more load-generator replicas.
- Aggregate per-query results into a unified test-run report.
- Persist reports to disk (JSON + CSV) with minimal processing overhead.

Data flow

1. Load generators POST metrics to `/api/addMetrics`.

2. Metrics are stored in a thread-safe map keyed by `loadGeneratorInstance`; the service counts received replicas.
3. When `load.generator.replicas` is reached, JSON/CSV export is triggered and a summary path is returned.
4. `ReportService` parses raw JSON responses, derives stats (avg/min/max for roundtrip and OpenSearch `took`), counts errors, and writes reports to `report.output.directory`.

The Metrics Reporter consists of the following core code components:

- `MetricsReporterApplication` (`metrics-reporter/.../MetricsReporterApplication.java`): Spring Boot entrypoint.
- `ReportController` (`.../controller/ReportController.java`): REST API (`/api/addMetrics`, `/api/health`); holds thread-safe storage and triggers report generation once all replicas report.
- `Metrics` (`.../dto/Metrics.java`): In-memory DTO for one load generator's batched metrics (aligned arrays of requestType/roundtrip/jsonResponse).
- `QueryResult` (`.../dto/QueryResult.java`): Per-query record with derived fields (roundtrip, opensearch took, hits, error flag).
- `TestRunReport` (`.../dto/TestRunReport.java`): Aggregate report containing statistics, totals, instances, and all `QueryResult` entries.
- `ReportService` (`.../service/ReportService.java`): Converts `Metrics` → `QueryResult`, calculates statistics, appends to JSON/CSV, manages report paths and initialization.

API contract

- `POST /api/addMetrics`: expects aligned arrays; validates presence of `requestType`, `roundtripMilSec`, `jsonResponse`, `loadGeneratorInstance`. On completion of expected replicas, returns paths to generated reports. Uses synchronized handler and `ConcurrentHashMap` + `AtomicInteger` for safety.
- `GET /api/health`: simple liveness check.

Report formats

- JSON (`report.json.filename`): pretty-printed, contains statistics (avg/min/max), totals, load-generator instances, and full query results.
- CSV (`report.csv.filename`): per-query rows (instance, request type, roundtrip, OpenSearch took, hits count, hasError, raw response).

Configuration (application.properties)

- `load.generator.replicas`: expected replica count before export (default 1).
- `report.output.directory`: default `./reports`.
- `report.json.filename`, `report.csv.filename`.
- `report.export.json.enabled`, `report.export.csv.enabled`.
- Logging: `logging.level.com.opensearchloadtester.metricsreporter`.

Error handling and validation

- Rejects missing required fields with HTTP 400.
- Marks queries as errors if response contains `error` or JSON parsing fails.
- Unknown fields are ignored; metrics arrays must stay aligned to avoid partial data.

Extensibility notes

- To add new derived metrics, extend `QueryResult` and `TestRunReport.Statistics`, then update `ReportService.calculateStatistics`.