Module Code
COM00055H

BSc and BEng Degrees 2024–25

Open Assessment

Department Computer Science

Module Engineering 2: Automated Software Engineering

Title Individual Coursework

Issued Monday 10 February 2025

Submission due 12 noon, Monday 19 May 2025

Feeedback and Marks due Monday 16 June 2025

Allocation of Marks: 50 marks for the data-intensive systems part, 50 marks for the application of model-driven engineering.

Instructions:

This is an anonymous assessment. Do **not** include your name, student number or any other identifying information in the submitted material (including metadata).

Your submission must include a report and an implementation. You must submit a single ZIP file containing your report in PDF format, and your implementation files. The report must use single-spaced lines using a sans-serif 11 point font (e.g. Arial), in a single-column layout. Submissions in a different format (e.g. a Microsoft Word document in a RAR archive) **will be penalised**.

Note the page limits: parts of answers that go beyond the page limit may not be marked. Use the IEEE referencing style: references must be listed at the end of the document and do not count towards the page limit.

The submission point has a **file size limit of 100MB**, which should be more than enough so long as you remove any temporary build files prior to submission (e.g. by running the Gradle `clean` task).

Any queries on this assessment should be addressed by email to Dr. Antonio Garcia-Dominguez at a.garcia-dominguez@york.ac.uk. Answers that apply to all students will be posted on the VLE.

All students should submit their answers through the appropriate Turnitin submission point in the Assessment area of the VLE site by 12 noon, Monday 19 May 2025. An assessment submitted after this deadline will be marked initially as if

it had been handed in on time, but the Board of Examiners will normally apply a lateness penalty.

Your attention is drawn to the section about Academic Misconduct in your Departmental Handbook.

Any queries you may have on this assessment should be posted on the Discussion Board on the Virtual Learning Environment (VLE) page for Engineering 2: Automated Software Engineering in the appropriate discussion area. **No questions will be answered after 17 May 2025**.

Note on Academic Integrity

This is an open assessment, and you are therefore permitted to refer to written and online materials to aid you in your answers. However, you must ensure that the work you submit is entirely your own, and for the whole time the assessment is live you must not:

- communicate with other students on the topic of this assessment.
- seek advice or contribution from any other third party, including proofreaders, friends, or family members.

We expect, and trust, that all our students will seek to maintain the integrity of the assessment, and of their award, through ensuring that these instructions are strictly followed. Where evidence of academic misconduct is evident this will be addressed in line with the Academic Misconduct Policy and if proven be penalised in line with the appropriate penalty table. Given the nature of these assessments, any collusion identified will normally be treated as cheating/breach of assessment regulations and penalised using the appropriate penalty table (see AM3.3. of the Academic Misconduct Policy).

*Please note: The publication of your (re)assessment submission online or elsewhere, or the sharing of these documents with others, is **NOT PERMITTED** until all students have completed this module assessment and reassessment (this will include any students with extensions to their reassessment date). We will notify you through the VLE at such time.*

1 The Exercise

Starting from the provided codebase (described in Section 2), you are to 1) complete the implementation and testing of the microservices of a food ordering app, and 2) implement a domain-specific modeling language to allow the marketing department to maintain the offers implemented by the system.

1.1 Part 1: Data-Intensive System

The food ordering app needs these two microservices:

- A Product Management (PM) microservice, with resources for listing, creating, updating, and deleting the products that can be ordered, calculating the unit and total prices of an order (while taking into account any relevant offers), and retrieving the number of daily orders for a given product.
- An Order Management (OM) microservice, with resources for listing, creating, updating, and deleting customers and their orders. When placing an order, the microservice only accepts the delivery address and the various quantities of each product to be ordered: the order details are sent to PM's pricing resource, and the returned unit and total prices are stored in OM's database.

In order for PM to maintain the number of daily orders for a given product, OM must produce Kafka records every time an order is placed. PM must be able to update its database by consuming these records, without having to interact with OM.

Each microservice must only access its own separate MariaDB database, in order to ensure it is separately deployable and upgradeable according to the microservice definition used in this module. The starting code includes the MariaDB SQL schemas of the databases to be used for PM and OM: these must not be changed.

The microservices must be implemented using Micronaut 4. **Microservices not implemented in Micronaut will receive a mark of 0**.

The system must be packaged as an orchestration of Docker containers (using Docker Compose), where each microservice uses its own Docker image. You must write unit tests for each separate microservice, as well as end-to-end tests covering the interactions between the microservices.

To keep this individual assessment manageable, we will assume some simplifications:

- You are not expected to develop any user interfaces for these microservices. Instead, you must use Micronaut's OpenAPI support to automatically produce web-based Swagger UI

clients from your controller classes.

- You are not expected to implement authentication or authorisation for the microservices.

1.2 Part 2: Application of Model-Driven Engineering

The offers in PM are to be maintained by staff from the marketing department. Due to the different skillset of the marketing staff, you are to develop a graphical modelling notation (using Eclipse Sirius) to allow them to maintain the offers without needing to know how PM is implemented.

The notation must allow users to catalogue the products available for purchase, using a hierarchy of categories as well as various tags. For instance, there could be a “Cakes” subcategory within “Sweets”, containing a “Large Chocolate Cake” product that would have the “chocolate”, “large”, and “cake” tags.

The notation must be expressive enough to define *offer rules* such as the following:

- “10% off from the first 10 orders of Bakewell Tarts in the day”
- “2 large pizzas for 10 pounds, with one free large ice cream”
- “2 pizzas for the price of 1”, but only if the above offer was not already applied.
- “10% discount on Christmas Day (60 GBP minimum order)”
- “5 pounds off on orders with chocolate cake (50 GBP minimum order)”, but only if the above Christmas Day offer does not apply.

These rules can be expressed as a combination of *conditions* for being activated (e.g. minimum order, or having a product with a certain set of tags), and *actions* when triggered (e.g. applying a certain type of discount).

In order to apply offers in a consistent order, your notation must support defining dependencies between rules. A rule can trigger another rule if the order matched its conditions (“if match”), if the order did not match its conditions (“if not match”), or “always”.

The developed Sirius-based graphical editor must support automated validation. It must check these properties:

- Product names must be unique.
- Percentage-based discounts must be between 0 and 100.
- There must be exactly one rule that is not triggered by any other (the *initial rule*). This ensures there is always a clear starting point.

- Each rule can only trigger at most one other rule for a given situation (“if match”, “if not match”, or “always”). This ensures that we always evaluate at most one rule after the current one, avoiding ambiguities.
- If a rule “always” triggers another rule, it cannot have “if match” nor “if not match” triggers.
- There must not be any direct or indirect dependency cycles between rules (e.g. rule A triggering rule B, which triggers rule C, which triggers rule A again).

You must write a model-to-text transformation that generates the Java code that will implement these rules in PM. You should consider approaches to clearly separate the generated code from any manually-written support code (e.g. use of class inheritance, or protected regions).

2 Starting code

The starting code is available as a Github template repository: please click on the previous link in this sentence, or follow the link in the “Assessment” section of the VLE. We recommend that you use the “Use this template” button on Github to create your own repository to work on this assessment. Please ensure that you maintain the integrity of the assessment by making your repository **private** and not sharing any of it during the whole time the assessment is live (including reassessments).

The starting code is divided into two main parts, which are described below, and which you will have to complete with your microservices and graphical modeling notation. You must follow this same folder structure for your submission, adding the PDF of your report. We also recommend that prior to submission, you test the ZIP to be uploaded by unpacking it and ensuring all the Gradle and Eclipse projects work as intended.

2.1 Part 1: Data-Intensive System

The `microservices` folder must contain all the code related to your microservices. Specifically, it must have the following contents:

- `product-management`: Gradle project to be completed with your code for PM.
- `order-management`: Gradle project to be completed with your code for OM.
- `end2end-tests`: Gradle project to be completed with your end-to-end tests.
- `compose.yml`: Docker Compose file which you must complete so it starts up all your containerised microservices together and their dependencies (e.g. their MariaDB databases and the shared Kafka cluster).

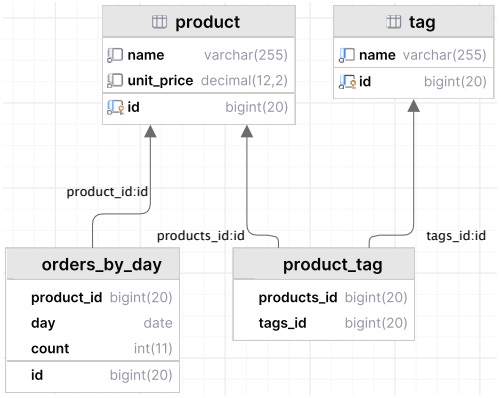

Figure 1: Tables of the Product Management database

- `run-tests.sh`: Bash script that will start the containerised version of your system and run tests on it. You should not have to make any changes to it.

The OM, PM, and end-to-end test projects must be buildable independently from each other: do not add any dependencies between them. Instead, you must use the OpenAPI descriptions produced by Micronaut for OM and PM in order to have OM invoke PM, and to invoke OM and PM from your end-to-end tests.

Both OM and PM already have the basic Gradle setup for a Micronaut project: you are expected to complete their implementation by writing all the necessary Java code. Their `src/main/resources/db/migration` folders already contain the SQL scripts needed to have Micronaut Flyway automatically set up their databases on startup. You should not make any changes to these SQL scripts: the database schemas for OM and PM are explained below.

IMPORTANT: before submission, make sure to remove all temporary files produced by your builds. You can do this by running the `clean` Gradle task on all your Gradle projects. Otherwise, you may have problems with file size limits during the upload.

2.1.1 Product Management Data Model

The starting code for PM includes SQL scripts defining the tables in Figure 1. The *product* table holds the unit price and unique name of each product. Products may have *tag*s associated to them, which have their own unique name. There is an *orders_by_day* table that keeps track of the number of orders that had a given product in a given day.

While not shown in the figure, you will notice that the SQL scripts also set up a *hibernate_sequence* sequence. This is required by the Hibernate object-relational mapping library we use in Micronaut Data. The Flyway tool used for automated database migration will also create its own table to keep track of any migrations that have been applied.

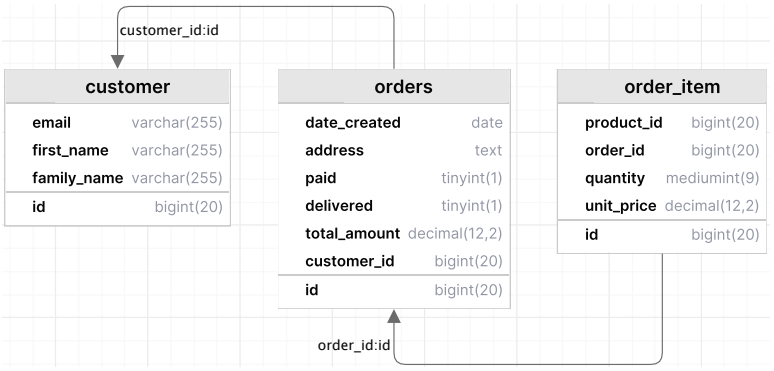

Figure 2: Tables of the Order Management database

2.1.2 Order Management Data Model

Likewise, the starting code for OM has the tables in Figure 2. There is a *customer* table with the name and contact information of the customer. An *orders* table contains the various orders from each customer, with their total amount (after offers), date of creation, delivery address, and payment/delivery statuses (true if paid/delivered, false otherwise). Finally, an *order_item* table lists the various items in each order: product ID (according to PM), quantity (note that we do not support fractional units), and unit prices.

It is important to note that this database does **not** store any product information: OM is expected to invoke PM to compute the total price of an order while applying any relevant offers.

2.2 Part 2: Application of Model-Driven Engineering

The starting code includes two folders dedicated to this part:

- `language` will need to be completed with the implementation of your graphical modelling notation, and must be divided into these folders:
  - `uk.ac.york.cs.eng2.offers`: definition of the metamodel with the abstract syntax of your notation, using Emfatic.
  - `uk.ac.york.cs.eng2.offers.{edit, editor}`: edit and editor plugins generated by EMF from your `.genmodel`.
  - `uk.ac.york.cs.eng2.offers.evl`: validation rules for your notation, implemented with EVL.
  - `uk.ac.york.cs.eng2.offers.generator`: model-to-text transformations for your notation, implemented with EGX and EGL.

  - `uk.ac.york.cs.eng2.offers.generator.dt`: Eclipse plugin that integrates the generator into the context menu of the Package Explorer. You should not have to change this.
  - `uk.ac.york.cs.eng2.offers.viewpoint`: Eclipse Sirius viewpoint that implements the graphical editor(s) for your notation.
- `model` must be an Eclipse project with your model of the offer rules listed in Section 1.2. It must contain at least:
  - You will need to add a `model.offers` file which uses your metamodel to express the offer rules, and which must be used to generate the Java code implementing those offers in PM.
  - `representations.aird`: Eclipse Sirius representations file, to be extended to add the diagrams relevant to your model.

The rest of the section describes some of the specifics of these folders. For conciseness, we will only use the last part of their folder name (e.g. `.offers` or `.evl`).

2.2.1 Metamodel

The `.offers` project contains a starting template for your metamodel in `model/offers.emf`. You must replace the rest of the file from `class Model` onwards with a metamodel of your design.

Emfatic will generate the `.ecore` file from the `.emf` file, and the `.genmodel` file should be usable as-is for generating the model, edit, and editor code.

2.2.2 Validation

A `.evl` project has already been set up for you so that any EVL rules in `evl/offers.evl` will be automatically used when validating with Sirius. You will need to write those EVL rules.

2.2.3 Code Generator

The `.generator` project has already been set up for you so that it is possible to right-click on an `.offers` file, select a target folder, and populate it by executing the EGX rules in `src/uk/ac/york/cs/eng2/offers/generator/main.egx` against it.

You should not need to change any of the Java code: you should focus on completing the EGX script and adding any necessary EGL templates.

2.2.4 Sirius Viewpoint Descriptors

The `.viewpoint` project is an Eclipse Sirius viewpoint descriptor project, with an empty `description/viewpoint.odesign` that you will have to complete with your Sirius-based graphical editor.

3 Questions

3.1 Part 1: Data-Intensive System (50 marks)

1 (10 marks) Architecture (max 2 pages)

Define the overall architecture using the C4 notation: include the context and container diagrams of the overall system, and the component diagram of either OM and PM. Explain how the system could scale up to more products, more customers, and higher customer activity (i.e. more orders per minute).

2 (20 marks) Microservices (max 2 pages)

Implement OM and PM using Micronaut 4. Each microservice must be independently deployable and scalable. Microservices must be able to run locally without requiring any cloud-based resources: any required persistence solutions must run locally.

Briefly report on how the implemented microservices meet the requirements in Section 1.1.

3 (5 marks) Containerisation (only code)

Produce Docker images for OM and PM using the built-in capabilities of Micronaut, and complete the Docker Compose file in the starting code so the `run-tests.sh` scripts will be able to start these microservices and all their dependencies, while following the good practices for production-readiness covered in the practicals.

4 (15 marks) Quality assurance (max 2 pages)

Perform unit tests for each microservice, and end-to-end tests across both microservices. Your unit tests must be executable through the Gradle `test` task without needing any other preparation, and your end-to-end tests must be executable through the provided `run-tests.sh` script.

Give a brief report on the actual tests, discussing how they were designed and implemented, and providing evidence of their correctness and completeness.

 Continued.

3.2 Part 2: Application of Model-Driven Engineering (50 marks)

5 (8 marks) Metamodel (max 2 pages)

Use Emfatic/Ecore to define a metamodel for the domain-specific language described in Section 1.2. Include a class diagram of the metamodel in your report, discuss the metamodel, state any assumptions you have made, and explain any alternative design decisions that you have considered and discounted.

6 (15 marks) Graphical concrete syntax (max 2 pages)

Use Eclipse Sirius to define and implement a graphical concrete syntax for your metamodel. Model the example rules in Section 1.2 in your language, provide a screenshot of the model in your concrete syntax, discuss and justify your syntax design and implementation decisions, and reflect on the strengths and weaknesses of the selected concrete syntax compared to alternatives.

7 (12 marks) Validation (max 1 page)

Use the Epsilon Validation Language to implement any validation constraints required in Section 1.2, which cannot be expressed in the metamodel itself.

Briefly explain the rationale and implementation of each constraint.

8 (15 marks) Model-to-text transformation (max 2 pages)

Use the Epsilon Generation Language to implement a model-to-text transformation that consumes a model that conforms to your DSML, and produces the code needed in PM to implement those rules. Discuss the model-to-text transformations and justify the organisation of the generated code.

END OF PAPER