Journal of Computer Science

Original Research Paper

GenMicro: A Tool for Generating Microservice Architectures with In-Depth Microservice Design

Willy Kengne Kungne, Georges-Edouard Kouamou and Paul Ayang

Department of Computer Engineering, National Advanced School of Engineering, University of Yaoundé I, Cameroon

Article history
Received: 30-08-2024
Revised: 11-12-2024
Accepted: 11-02-2025

Corresponding Author:
Willy Kengne Kungne
Department of Computer Engineering, University of Yaoundé I, Cameroon
Email: w.kungne@gmail.com

Abstract: MicroService Architecture (MSA) has become an increasingly popular architectural style for distributed and service-based systems. Despite the various facilities offered by existing frameworks, setting up a microservice architecture remains challenging for developers. Also, configuring all the microservices is time-consuming. Several tools have been proposed for generating microservice architectures. For the most part, these tools focus on the basic configuration aspects, leaving the business aspects of each microservice to the developers. Thus, a question arises: How can business elements be integrated into the definition of formalism in microservices architecture alongside the configuration aspects? This study proposes a tool named *GenMicro*, which is based on a detailed design. It takes into account various elements such as business components, entities, dependencies, or configurations encapsulated within nodes representing microservices, utilizing a model-driven engineering approach to transform them into code models for code generation. The tool has three components: A graphical editor for architectural representation and internal description of each microservice, an intermediate transformation engine to transform the graphical elements into a code model, and a module to refine the code according to the microservice architecture. The ready-to-use Java code generated by *GenMicro* is compliant with the Spring Cloud Netflix Framework and is deployment-ready.

Keywords: Microservice Architecture, Model-Driven Engineering, Business Domain, Domain-Specific Languages, Java Code

Introduction

Microservice Architecture (MSA) is an increasingly favored architectural style for distributed and service-based systems (Sorgalla *et al.*, 2018; Dragoni *et al.*, 2017; Mazzara *et al.*, 2021). It utilizes the service concept as the fundamental building block for a system's architecture. A microservice is a cohesive and independent process that interacts via messages (Dragoni *et al.*, 2017; Pahl and Jamshidi, 2016; Mazzara *et al.*, 2021; Nadareishvili *et al.*, 2016; Neuman, 2015). This architecture represents an emerging development paradigm in which software is constructed by composing autonomous entities known as microservices (Neuman, 2015; Thönes, 2015; Shadija *et al.*, 2017). Microservices communicate using Representational State Transfer (REST) (Haupt *et al.*, 2014) or Message Queue (MQ) protocols.

To simplify the implementation complexity of MSA, Model-Driven Engineering (MDE) is commonly employed. MDE promotes the extensive use of models throughout the software development process, leading to the automated generation of the final application (Whittle *et al.*, 2014). The benefits of MDE include enabling informed design decisions, ensuring that design teams understand what is being developed, where to make partitioning choices, and how the system will be built. It effectively masks the great complexity of the various configurations to be implemented. Moreover, models can be defined using general-purpose modeling languages like UML (Eriksson *et al.*, 2003), although Domain Specific Languages (DSL) (Mernik *et al.*, 2005) are often used for specific, well-defined domains. Tools based on UML or DSL, for example, reduce the time and errors involved in defining MSA.

The tools used for implementing microservices architectures must consider the following key factors:

(i) Configuration of each microservice: Define the role of the microservice along with its input and output parameters

Science Publications

© 2025 Willy Kengne Kungne, Georges-Edouard Kouamou and Paul Ayang. This open-access article is distributed under a Creative Commons Attribution (CC-BY) 4.0 license.

(ii) Communication between microservices: The Discovery server will discover each microservice. Each microservice has a route configured in the APIGateway, allowing one microservice to call another within the architecture
(iii) The internal structure of each microservice: This involves incorporating the business aspects of each microservice to understand its functionalities.
(iv) Graphical definition of architectural elements: Graphical tools enable developers to easily define or modify architectural elements, unlike text-based tools, where language syntax must be strictly followed. This syntax can often be difficult to master, whereas a graphical interface simplifies the developers' work

Existing tools such as AjiL (Sorgalla *et al*., 2018), JHipster (Raible, 2016), MAGMA (Wizenty *et al*., 2017), MicroBuilder (Terzić *et al*., 2018), Sliceable Monolith (Montesi *et al*., 2021), Microservice DSL (MDSL) (Zimmermann *et al*., 2022), Magic (Bucchiarone *et al*., 2023) and Silvera (Suljkanović *et al*., 2022) primarily focus on the first two points, leaving the third to the developers. Regarding the fourth point, tools such as AjiL offer graphical interfaces for defining the architecture's basic configuration.

This study proposes a new Domain-Specific Language (DSL) for the automatic generation of Microservices Architectures (MSAs). The proposed DSL emphasizes the business aspect of a microservice in addition to its basic configuration. It employs detailed design elements such as class diagrams, component diagrams, and deployment diagrams to generate the MSA of the application, where each node represents a microservice.

The rest of the paper is organized as follows: The second section covers the literature review, while the third section presents *GenMicro* modeling. The fourth section introduces the developed tool, and the fifth section provides a case study of a shopping payment application. The sixth section includes a discussion, and we conclude with a summary and highlight future perspectives.

Background

Various definitions have been proposed for a microservice, as presented in the previous section. These definitions emphasize some level of independence, limited scope, and interoperability. It is also important to view a microservice within the context of an existing system. From these definitions and its architecture, microservices have several characteristics (Dragoni *et al*., 2017):

(i) Bounded context: Related functionalities are combined into a single business capability, which is then implemented as a service
(ii) Independence: Each microservice operates autonomously from others
(iii) Flexibility: A system can adapt to the ever-changing business environment and support necessary modifications to remain competitive in the market
(iv) Modularity: A system is composed of isolated components, with each component contributing to the overall system behavior rather than having a single component that offers full functionality
(v) Evolution: A system should remain maintainable while constantly evolving and adding new features

In summary, MSAs consist of independently evolving microservices that collaborate to implement a business application. In order to hide the complexity of configuring and implementing MSAs, Model Driven Architecture (MDA) can be used. MDA is a specific proposition for implementing MDE proposed by the Object Management Group (OMG). It describes an approach based on metamodels, abstract models (Platform-Independent Models, PIM), and more specific models (Platform Specific Model, PSM) that can be used to generate source code.

MDA includes a set of modeling and model transformation techniques standardized by the OMG (Kleppe *et al*., 2003; Blanc and Salvatori, 2011). This approach promotes the use of models throughout various phases of an application's development cycle. The fundamental principle of MDA is to develop Platform-Independent Models (PIM) and transform them into Platform-Specific Models (PSM) for concrete implementation of the system. Transformations between PIM and PSM are typically carried out using automated tools. Fig. (1) illustrates the different phases of the approach. The transformation tools are compatible with the OMG standard known as Query, View, and Transformation (QVT). In this context, we propose a Domain-Specific Language (DSL) to model the graphical elements of our MSA. Model transformations will be used to refine the models designed in the DSL into code.

A DSL is a high-level software implementation language that supports concepts and abstractions related to a particular (application) domain (Lämmel *et al*., 2008). The aim of DSLs is to enhance the productivity of software engineers by abstracting away low-level boilerplate codes. In the next section, we present the DSL tools, the majority of which use Model Driven Engineering (MDE) to generate the MSAs.

Related Works

In recent years, various solutions have been proposed to facilitate the generation of MSAs. In the following paragraphs, we present a few of these tools and compare them based on ease of use and the extent to which business aspects are considered in the architecture description.

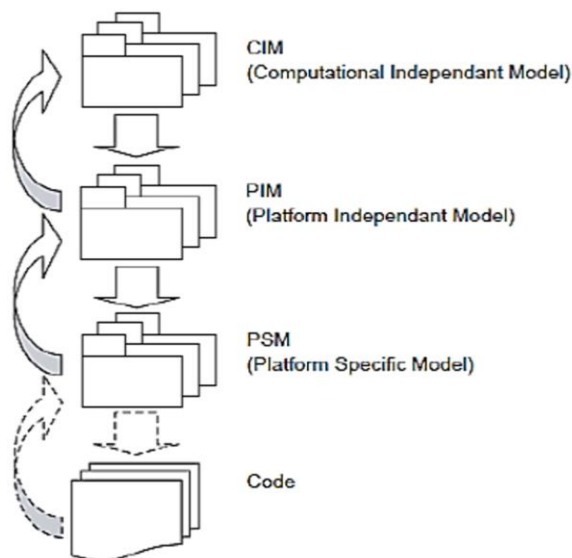


Fig. 1: Overview of the MDA approach (Blanc and Salvatori, 2011)

JHipster (Raible, 2016) is a well-established tool for generating Web form-based MSAs and includes a textual modeling language for defining entities. JHipster's objective is to provide a fully functional architecture with front-end and back-end components. Technologically, it depends on the Spring Framework for the back-end and the various technologies available for front-end implementation, such as Angular. JHipster does text-based entity modeling.

MAGMA (Wizenty *et al.*, 2017) is a build management tool based on Maven's archetype mechanism capable of creating microservice foundations based on predefined service models. However, due to its Maven dependency, microservices are described textually, and it lacks the ability to interlink individually generated microservices.

AjiL (Sorgalla *et al.*, 2018) includes a full-fledged graphical DSML dedicated to MSA, offering the possibility of modeling a complete system base. AjiL does not aim to provide fully functional back-end and front-end components but rather aids developers in avoiding tedious and redundant coding when creating their own customized MSA. Notably, all modeled service interfaces are generated as REST controllers exposing Create Read Update Delete (CRUD) operations. Ajil focuses on generating the code base while only modeling the basic configuration of the MSA.

MicroBuilder (Terzić *et al.*, 2018) comprises MicroDSL & MicroGenerator, generating code for a REST MSA using a model-based approach. Its aim is to simplify the development of microservice-based applications by handling complex tasks such as microservice architecture configuration, load balancing, automatic discovery and registration of microservices, thus reducing development time by eliminating redundant code templates. However, it emphasizes textual specification without detailing the internal structure of a microservice.

Sliceable Monolith (Montesi *et al.*, 2021) advocates first designing a monolithic architecture and then transforming it into an MSA using the Jolie language to specify the architecture. The approach differs from the other tools in that it starts with a monolithic architecture and evolves towards microservices. However, Jolie, being text-based, might require a considerable amount of time to grasp fully.

Microservice DSL (MDSL) (Zimmermann *et al.*, 2022) includes textual specifications based on Jolie. According to its author, the language is intended for contexts where a suite of microservices, along with their various communication protocols (e.g., HTTP, message queuing), need to be described. Additionally, the DSL can be used to represent their subsequent request and response messages, as well as the interface endpoint. Nevertheless, MDSL serves as a tool exclusively designed for specifying microservices rather than for direct development. The code produced in either Java or Jolie programming languages (Montesi *et al.*, 2014) serves as a template for the specified service, requiring developers to implement the actual functionality.

Magic (Bucchiarone *et al.*, 2023) is a DSL framework for implementing language-independent microservices-based Web applications. The framework can be used to specify and deploy microservices-based software applications end-to-end on Docker containers, which can then be used like any other application on the Internet. Magic focuses on the front end rather than the back end, and its definition is textual.

Silvera (Suljkanović *et al.*, 2022) aims to fulfill the following criteria: (i) The language is easy to use for both domain experts and beginners, (ii) It supports well-known MSA design patterns as first-class concepts, (iii) It supports heterogeneous technology stacks via an extensible code generator framework, (iv) It provides automatic generation of OpenAPI architecture diagrams and documentation, and (v) It uses metrics tailored to microservices to evaluate the architecture of the designed system. However, via the SilveraDSL module, the description of microservices is textual and does not take into account the business aspect.

As a general remark, the tools reviewed are mostly text-based, a feature that adds to the challenge of mastering their usage. In contrast, the graphical tools model the basic configuration, leaving the internal modeling of each microservice to developers. The following section delves into the definition of *GenMicro*, which aims to enhance the language for microservices architecture with a focus on the business aspect.

DOI: 10.3844/jcssp.2025.729.740

Materials and Methods

GenMicro is built using a series of steps based on the MDA methodology:

- Step 1: Formalisation (PIM) & Constraints
- Step 2: Codes Model (PSM)
- Step 3: Transformation Rules (PIM to PSM) & Refinement (PSM to Java Codes)

Step 1: Formalisation (PIM) & Constraints

PIM allows to build models that are not linked to the underlying technology. We offer models that represent class diagrams, business components, and boxes to represent elements such as configurations and dependencies, as well as Microservices. The EMF-based abstract syntax tree in Fig. (2) illustrates an application based on the MSA, representing the PIM model.

An application consists of a set of microservices, each containing configurations and dependencies. Three types of microservices have been identified:

(i) Discovery: A microservices-based application generally runs in virtualized or containerized environments. The number of instances of a service and their locations change dynamically. We need to know where these instances are and their names to enable requests to reach the target microservice. This is where the microservice Discovery is important. Discovery acts as a registry in which the addresses of all instances are tracked. So there's an implicit link between all the microservices and the Discovery microservice
(ii) APIGateway: An APIGateway is a conductor that organizes the requests processed by the MSA to create a simplified user experience. An APIGateway is set up in front of the microservices and becomes the entry point for each new request executed by the application. It communicates with all other business microservices
(iii) BusinessMicro: A BusinessMicro represents the business of each microservice. The internal architecture of a microservice is a layered architecture where one can represent the controllers via the SetService class and the business via the Business class. This latter class contains a set of methods that implement the services that will be exposed in the controllers. The final layer is the database model, represented by the Entity, RelationShip, IdClass, and Attribute classes. These different classes are derived from the work of Kouamou and Kungne (2017), which propose an approach to generating layered architectures.

Structural constraints within the meta-model have been established to ensure specific criteria, including verifying the definition of class names, preventing links from connecting entities across different microservices and enforcing that connections between a method and an entity are contained within the same microservice. Only one APIGateway and Discovery are allowed.

In Fig. (2), the Sequence class is used to describe a set of instructions implemented within a method, introducing the dynamic aspect by describing the method's behaviour. A sequence is linked to a method and possibly to an entity. It contains the following properties:

- *instructionType*: This indicates the type of the instruction which can take several values:
    - *CREATE* and *UPDATE:* Used to create or modify columns within the entity; the column(s) to be added/updated are in *attNames*
    - *CHECK:* Used to verify one or more columns in the database; the column(s) to be checked are in *attsName*
    - *DELETE:* Used to delete a line in the entity
    - *API:* Used to call an external API
- *attsName*: Contains the list of columns affected by the instruction type
- *seqNum*: This very important field specifies the sequence number, which will allow instructions to be sequenced according to their sequence number
- *apiRequest*: This field is used when the value of *instructionType* is set to API to specify the request to be implemented
- *apiResponse*: This field is used when the *instructionType* value is set to API to specify the properties of the returned object

Step 2: Codes Model (PSM)

To obtain a specific model, the execution platform(s) must be chosen (several platforms can be used to implement the same model). The runtime characteristics and configuration information that have been defined generically are converted to take account of the platform specifics of MSA. For example.

BusinessMicro, *APIGateway* & *Discovery* have been merged into Microservice, while Component and SetService have been merged into BusinessClass. Figure (3) shows the code model.

The code model adds technological elements to the PIM model (Fig. 2). Given that the target model (PSM) is object-oriented, the code is made up of a set of classes, and we have aggregated concepts such as *SetService*, *Component* & *Entity* into classes in a multi-layered structure, the main layers of which are *controller*, *business* and *dao*.

DOI: 10.3844/jcssp.2025.729.740

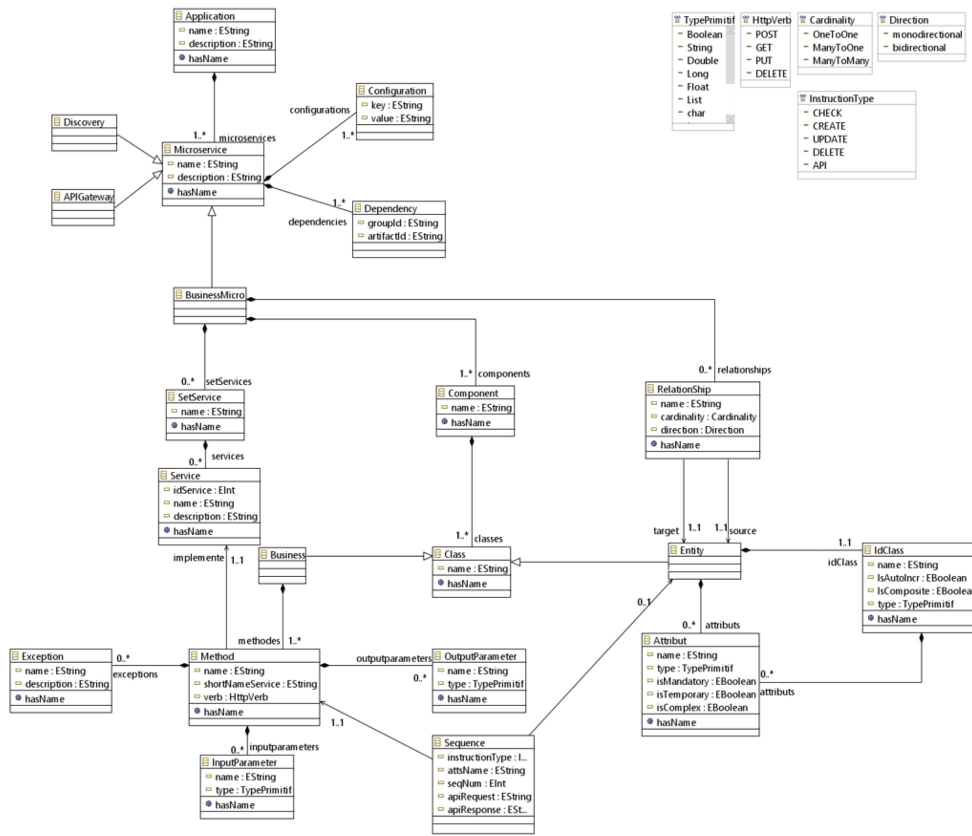


Fig. 2: Formalism for describing an MSA used in *GenMicro*

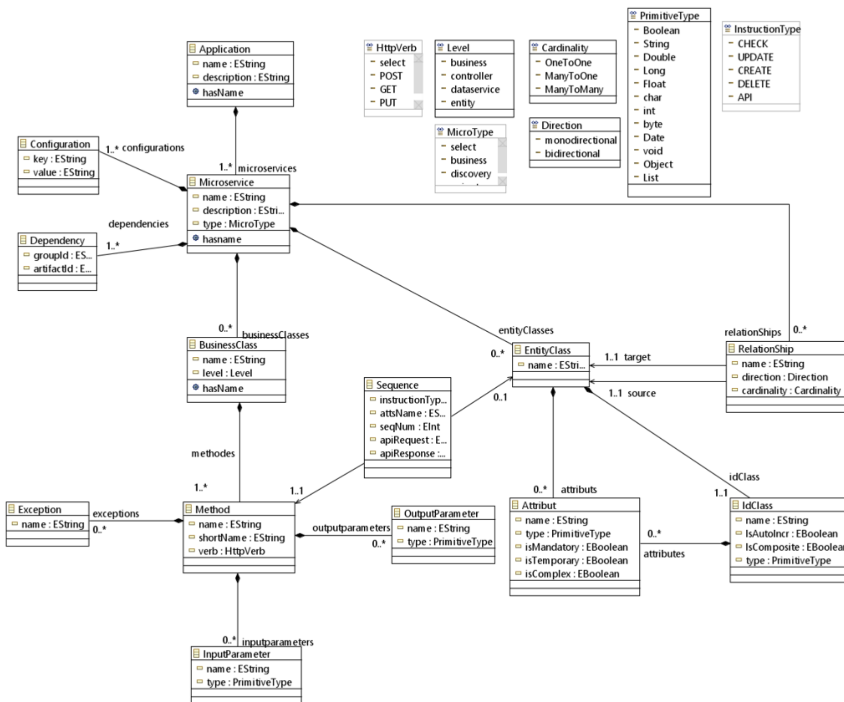


Fig. 3: Metamodel representing the code to be generated

With regard to the interpretation of the *Sequences* of instructions to be generated within a method, the user can specify the steps to be followed:

- If the *instructionType* is *CHECK*, a parameterized query (*findBy*) is created in the repository class and called inside the method
- If the *instructionType* is *CREATE*, a parameterised query is defined if it does not already exist. This query is then called to check if the object exists; the associated entity is instantiated, its properties are defined and the insert function is called.
- If the *instructionType* is *UPDATE*, a parameterised query is defined if it does not exist and invoked. The different properties are defined, and the update function is called.
- If the *instructionType* is *API*, the *apiRequest* and *apiResponse* attributes are used respectively to call a remote REST web service and build its response

The transformation rules are presented in the next subsection.

Step 3: Transformation Rules & Refinement

Table (1) presents a non-exhaustive list of transformation rules used to transfer concepts from the source meta-model (MSA description formalism) to the target meta-model (code model).

Table 1: Transformation rules

Source	Target	Comments
Application	Application	ApplicationToApplication: transforms the name, description, and set of microservices. Three types of microservices will be obtained
BusinessMicro	Microservice	businessMicro2Microservic: Transforms the name, description, #business microservice, configurations, dependencies, entities, and businessClass into the target business microservice
ApiGateway	Microservice	*apiGateway2Microservice*: transforms the name, description, *#apigateway* microservice type, configurations and dependencies into the target *#apigateway* microservice type
Discovery	Microservice	*discovery2Microservice*: Transforms the name, description, *#discovery* microservice, configurations and dependencies into the target *#discovery* microservice
Dependency	Dependency	*dependency2Dependency*: Transforms *groupId* & *artifactId* into their equivalents in the codes model
Configuration	Configuration	*configuration2Configuration*: Transforms key & value into their equivalents in the codes model
RelationShip	RelationShip	*relationShip2RelationShip:* Transforms the properties (*name*, *cardinality*, *direction*, *source*, and *target*) of a relationship into a code model that can be easily interpreted in the target language
Business	BusinessClass	*Business2BusinessClass:* Transforms business elements such as class names, attributes, and methods into a code model
Method	Method	*Method2Method:* Transforms the elements of a method, such as a name, parameters as inputs, parameters as outputs, and exceptions, into a code model

In the next section, we implement these rules using the ATLAS Transformation Language (ATL) language (Jouault *et al*., 2008).

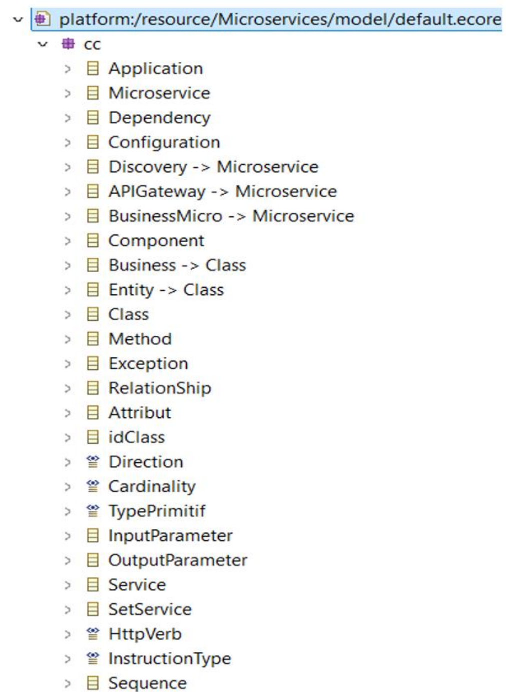
Refinement involves leveraging the experience gained over many years of software development. We introduce technological elements from the code model, transforming each element into classes or packages according to the components of the Spring Cloud Netflix framework.

Results and Discussion

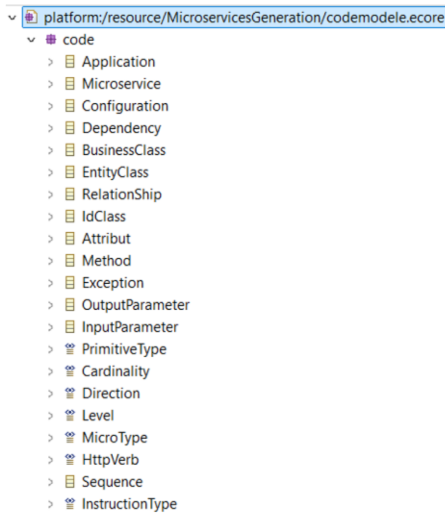
To facilitate the generation of MSA from analysis and design elements, the proposed tools are built as Eclipse pluginsplugins using the Eclipse Modeling Framework (EMF), the Graphical Modeling Framework (GMF), and ATL software.

Framework Building Environment

EMF is a modeling framework that involves code generation from a data model. It is a Java implementation of a subset of the OMG Meta-Object Facility (MOF) standard. To avoid ambiguities with MOF, EMF models conform to the eCORE meta-model. We used EMF to build the formalism of our models. Figures (4a-b) show the two eCORE meta-models we have developed. However, EMF does not offer graphical modeling tools, which is why GMF is also used. GMF is a framework for creating a graphical editor from a data model based on the Eclipse platform. This tool is composed of EMF and Graphical Editor Framework (GEF), the latter consisting of two parts:



(a)



(b)

Fig. 4: (a) eCore for microservices description; (b) eCore for codes model

- Graphic definition model: Represented by the *.gmfgraph* file extension, this model is used to specify the graphic elements of the model. Figure (5) shows the various graphical nodes defined.
- Tool definition model: Represented by the *.gmftool* file extension, this model is used to specify palette elements. Users can drag and drop palette elements to add new graphical elements, which can be a node (Entity, Component, Method, etc.) or a relationship (relationship between two entities or a sequence between a method and an entity). Figure (6) shows the defined palette elements.

To link the EMF models to GEF, GMF assembles a file with the *gmfmap* extension via the mapping model. Figure (7) defines the layout of the various compartments in order to define the graphical elements.

Each element of the graphical definition model is assigned a node and an action, along with the corresponding data model class. After this step, a new *.gmfgen* file can be generated to consolidate all project information. ATL allows you to specify the rules for transforming service models and logical views into the implementation view. Figure (8) illustrates 3 of the 16 rules defined.

These various elements enable us to define our graphical editor, with the main interface shown in Fig. (9). The central view of the interface shows the elements of the architecture being modeled, including nodes and relationships, whose properties can be consulted. The palette elements on the left can be dragged and dropped onto the central view, and no errors will occur as long as constraints are respected.

In the next subsection, we will implement an example of a simplified e-commerce application using our tool.

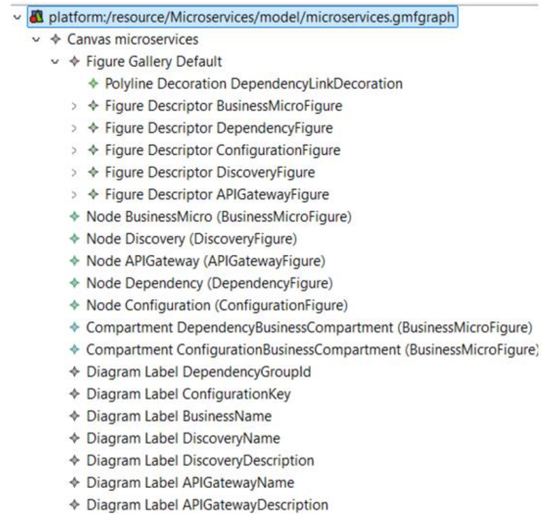


Fig. 5: Some nodes implemented

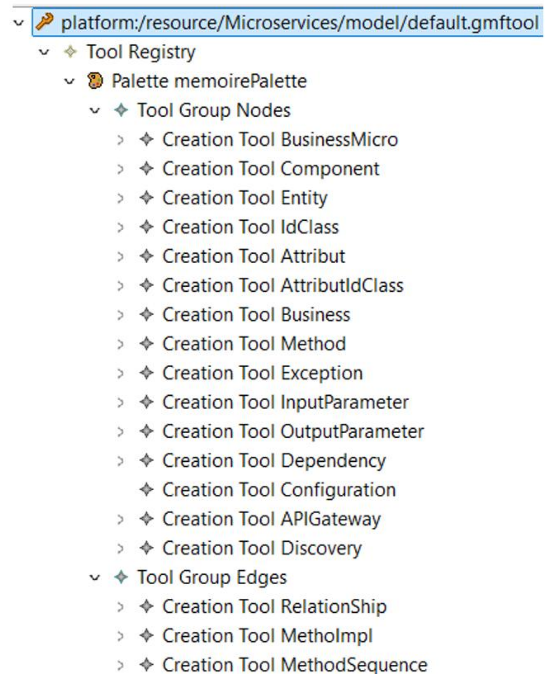


Fig. 6: Palette elements

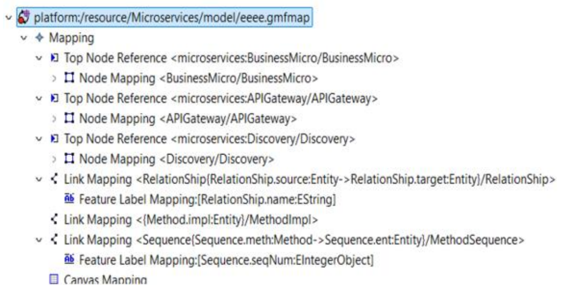


Fig. 7: Elements of the map

DOI: 10.3844/jcssp.2025.729.740

```
rule ApplicationToApplication {
    from e1 : cc!Application
    to out : code!Application (
        name <- e1.name,
        description <- e1.description,
        microservices <- Set{ cc!Microservice.allInstances() ->
                                collect (e | thisModule.resolveTemp(e, 'micro2')),
                                cc!Microservice.allInstances() ->
                                collect (e | thisModule.resolveTemp(e, 'micro1')),
                                cc!Microservice.allInstances() ->
                                collect (e | thisModule.resolveTemp(e, 'micro'))
                            }
    )
}
rule businessMicro2Microservice {
    from busMicro: cc!BusinessMicro
    to micro3 : code!Microservice (
        name <- busMicro.name,
        description <- busMicro.description,
        type <- #business,
        configurations <- thisModule.configurationOfMicroservice(busMicro),
        dependencies <- thisModule.dependencyOfMicroservice(busMicro),
        relationShips <- cc!RelationShip.allInstances() ->
                                collect (e | thisModule.resolveTemp(e, 'outrel')),
        businessClasses <- cc!Business.allInstances() ->
                                collect (e | thisModule.resolveTemp(e, 'bussClass2')),
        entityClasses  <- cc!Entity.allInstances() ->
                                collect (e | thisModule.resolveTemp(e, 'outent')),
        application <- busMicro.application
    )
}
rule apiGateway2Microservice {
    from api: cc!APIGateway
    to micro2 : code!Microservice (
        name <- api.name,
        description <- api.description,
        type <- #apigateway,
        configurations <- thisModule.configurationOfMicroservice(api),
        dependencies <- thisModule.dependencyOfMicroservice(api)
```

Fig. 8: Transformation rules implemented

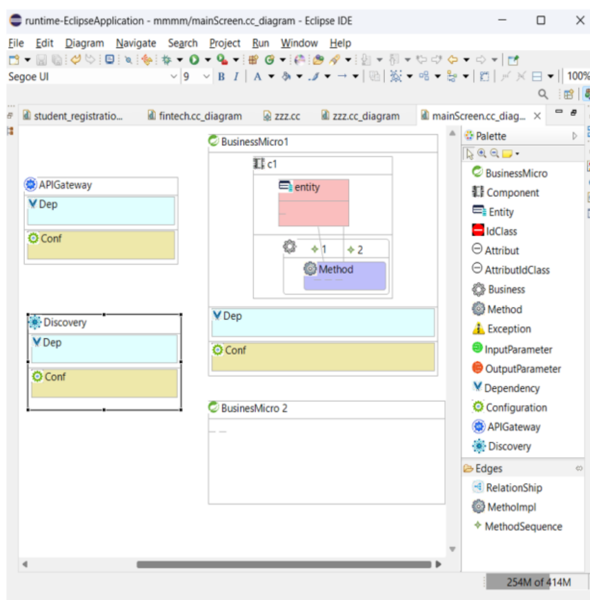


Fig. 9: Main Interface of the developed plugin

Case Study

This subsection presents a case study of an application for payment by transfer after product purchases. The application is divided into microservices for flexibility or scalability. Figure (10) shows a simplified view of the architecture, which includes three business services: *MSUser* for user management and authentication, *MSShop* for shopping cart management, and *MSBank* for managing bank transfer payments. In the next sub-section, we detail the functionality of each microservice and how they are designed using *GenMicro*.

Realization

Figure (11) shows the model of our case study, extracted from a *Fintech* Project. Using the Agile SCRUM method, three business microservices, along with the microservices for API Gateway and Discovery, were designed after a few sprints. Each microservice was designed using *GenMicro*'s graphical interface, which implicitly generates all CRUD methods for the defined entities and interprets personalised business methods. For example, in the *OrderManager* business component, the *addProduct* method checks that both *orderId* and *productId* exist before adding the product to the order by inserting a row in the *Contain* table. Sequences 1, 2, and 3, which are linked to *addProduct*, define this situation. Similar interpretations are applied to all the other personalized methods. In summary, the following microservices have been generated using *GenMicro*:

- A microservice for user management and authentication (*MSUser*): This microservice is used to manage security, which is essential in such a system. The business related to token creation and validation is implicitly generated and linked to the entity containing the login and password. The entities linked to the management of roles and privileges have been designed and associated with the business components. In Fig. (11), *AuthorizationManager* implements business services such as *listAuthOfProfile*, which lists a profile's authorizations and the redefinition of the save function (*saveAuth*). For each of these methods, the sequence of operations is defined.
- A microservice for managing the shopping basket (*MSShop*): This microservice is used to manage the orders of a bank user identified by their (mobile) telephone number. Customized methods such as adding a product to an order or listing the products in an order have been defined.
- A microservice for managing payments by transfer: For this microservice, the Account, Customer, and Transaction type entities have been designed. A customer can have multiple accounts each linked to the same customer. Similarly, a customer can carry out several types of operations. Business methods such as calculating fees or transferring money from a customer account to a merchant account have been specified.
- Microservices linked purely to the architecture configuration, such as API Gateway & Discovery, have been generated.

DOI: 10.3844/jcssp.2025.729.740

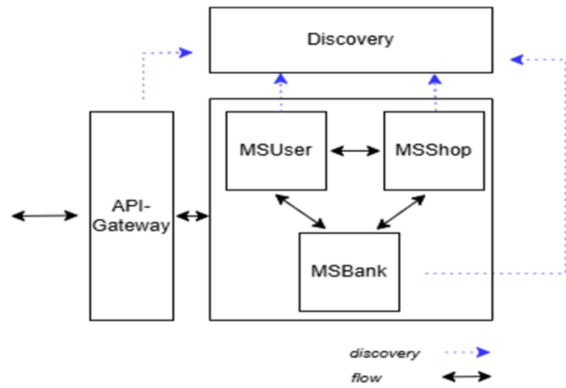


Fig. 10: Microservices architecture of the application

Model Transformation and Source Codes

The model in Fig. (11) is transformed using the rules in Fig. (8) and finally refined to obtain the code shown in Fig. (12). These codes are ready to be deployed in docker containers. For each microservice, a layered architecture was generated, including entities, repositories, businesses, and controllers. The DockerFile (for each microservice) and docker-compose.yml files were also generated. Once the microservices architecture has been generated, DevOps practices (Bass *et al*., 2015) can be used to manage the deployment of the microservices. Apart from the CRUD methods that are automatically generated, the user can design their own methods and link them to the entities as shown. This last aspect makes it possible to generate as much code as possible.

Genmicro not only makes it easier for developers to configure the microservices architecture but also helps them write the code for each microservice. For each microservice, 100% of the code has been generated. These codes are ready to be deployed. However, *GenMicro* should integrate the complete semantics of UML interaction diagrams instead of just providing sequences of instructions on the entities or API calls. GenMicro aims to assist developers in the realization of their projects. Based on the conceptual elements, the tool automatically generates the code to ensure that the software produced aligns with the designed requirements. Once the developers have understood the customer's requirements, GenMicro allows them to model (by dragging and dropping elements from the palette) the business elements (entities, business components …) and to save a lot of time by generating code. When conceptual elements are updated, the user can regenerate the code. The tool will take updates into account.

Discussion

Several tools have focused on MSA generation, as outlined in the introduction. These tools were revisited and discussed on the basis of their properties, which include: Basic configuration, graphical/textual nature, CRUD generation, and Business operation modeling. Table (2) provides a detailed comparison with existing tools based on the above properties.

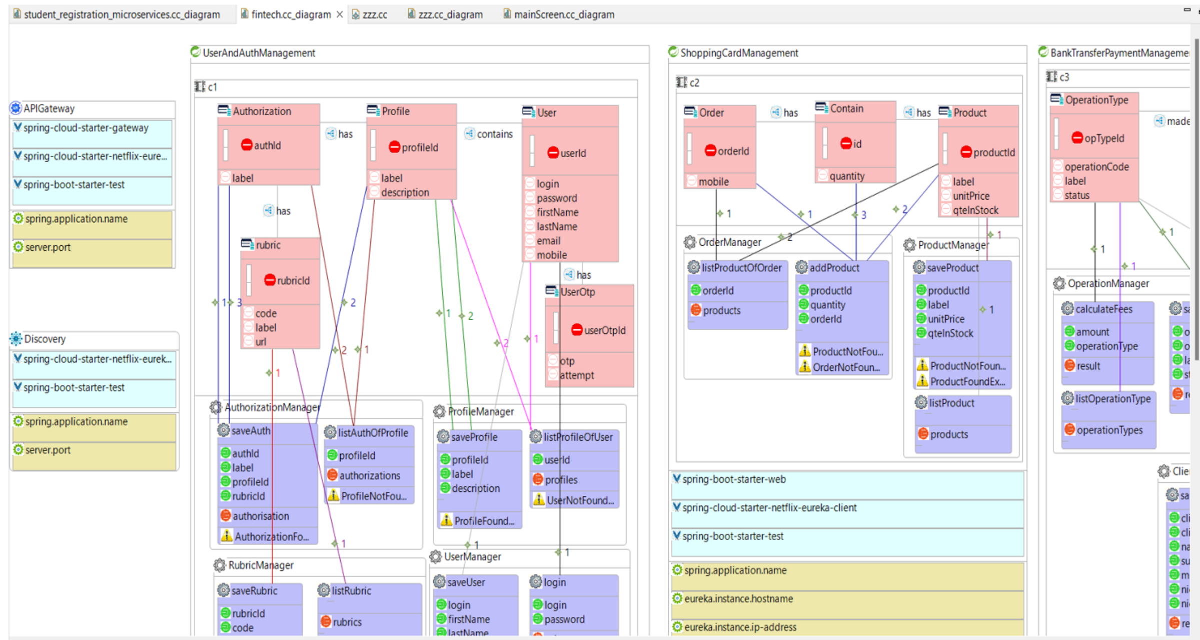


Fig. 11: Application modeling with *GenMicro*

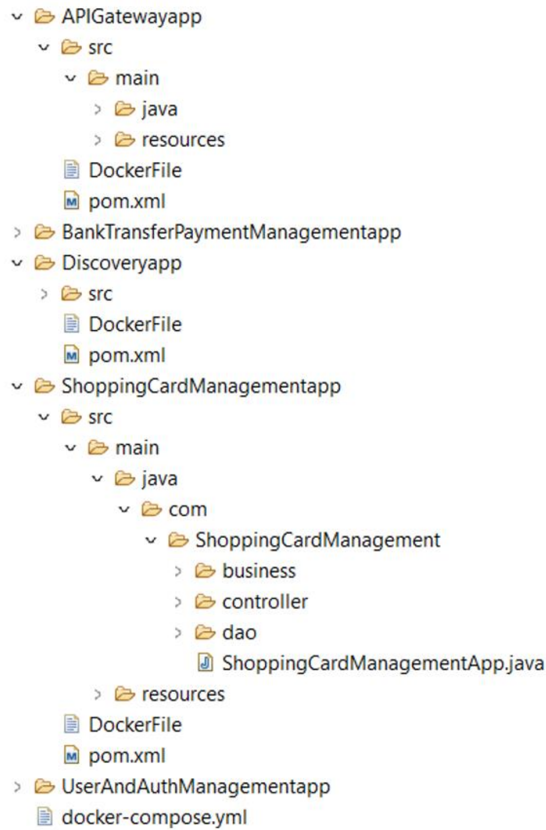


Fig. 12: Generated codes

Table 2: Comparison of *GenMicro* with existing tools

Existing tools	Basic config	Graphics	CRUD generation	Business operation modeling
JHipster	✓	×	✓	×
MAGMA	✓	×	×	×
AjiL	✓	✓	✓	×
Sliceable Monolith	✓	×	×	×
Microservice DSL	✓	×	✓	×
Magic	✓	×	×	×
Silvera	✓	×	×	×
GenMicro	✓	✓	✓	✓

While all the tools considered take into account the basic configuration of the various microservices, most do not address the internal modeling of microservices, which is one of the key elements of the tool presented in this study.

Graphical tools are generally easier to learn than textual ones, which can be more difficult to master. JHipster offers a textual language based on entities, while MAGMA also offers a textual language based on Maven. Sliceable Monolith and Microservice DSL (MDSL) are based on Jolie, a text-based approach for specifying Microservices architecture. Similarly, Silvera follows a text-based approach. In contrast, this study introduces a graphical editor tailored to model the internal structure of each microservice.

AjiL is a specialized graphical tool designed exclusively for Microservices Architecture (MSA), enabling users to model the basic elements of the system. While it does not emphasize the business aspect, all service interfaces modeled in AjiL are automatically generated into REST controllers that expose Create, Read, Update, and Delete (CRUD) operations, similar to the approach in Microservice DSL, where each Microservice exposes CRUD operations for interacting with specific datasets. JHipster can also be used to generate the CRUD elements of a modeled application. However, none of these tools can fully describe all data via diagrams or incorporate business operations comprehensively.

Conclusion

In this study, we propose a new tool for generating MSAs from system analysis and design elements. In addition to the basic configuration elements, we have integrated business domain elements into the tool. The core idea is to consider the internal structure of each microservice when generating the MSA. Using MDA, we proposed a suite of meta-models describing the MSA and the internal structure of each microservice. We used ATL to automate the generation of the intermediate model and its refinement into code targeting Java alongside the Spring Cloud Netflix Framework. The strength of the proposed tool lies in its consideration of the internal architecture of each microservice, which consistently aims to simplify developers' tasks. By taking as input a graphical representation of the class diagram, business components, and nodes representing each microservice, the tool converts them into code, thereby reducing application development costs. Some thought has been given to describing the dynamics of business methods. We have proposed a simple representation of the interactions between a method belonging to a business component and an entity, as well as the calling of external APIs. In future work, we intend to improve this representation by integrating all the UML semantics of sequence or collaboration diagrams. Despite the generation of DockerFile and docker-compose, an important point for future consideration is the integration of other scripts (Burns *et al.*, 2022) during the refinement process to facilitate the continuous deployment of the generated architecture.

Acknowledgment

We would like to thank all the reviewers of this study.

Funding Information

This research received no external funding.

DOI: 10.3844/jcssp.2025.729.740

Author's Contributions

Willy Kengne Kungne: Designed the tool, defined the methodology, developed the software and wrote the various versions of the manuscript.

Georges-Edouard Kouamou: Proposed the methodology and supervised the writing of the manuscript.

Paul Ayang: Contributed to the development of the software.

All authors have read and agreed to the published version of the manuscript.

Ethics

The authors affirm that this article is an original work and has not been previously published elsewhere.

References

Bass, L., Weber, I., & Zhu, Liming. (2015). *DevOps: A software architect's perspective*.

Blanc, X., & Salvatori, O. (2011). *MDA in action: Model-driven software engineering*. Editions Eyrolles.

Bucchiarone, A., Ciumedean, C., Soysal, K., Dragoni, N., & Pech, V. (2023). Magic: a DSL Framework for Implementing Language Agnostic Microservice-based Web Applications. *The Journal of Object Technology*, *22*(1), 1–21. https://doi.org/10.5381/jot.2023.22.1.a2

Burns, B., Beda, J., Hightower, K., & Evenson, L. (2022). *Kubernetes: Up and Running: Dive into the Future of Infrastructure*.

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. (2017). Microservices: Yesterday, Today and Tomorrow. *Present and Ulterior Software Engineering*, 195–216. https://doi.org/10.1007/978-3-319-67425-4_12

Eriksson, H.-E., Penker, Magnus, Lyons, B., & Fado, D. (2003). *UML 2 toolkit*.

Haupt, F., Fischer, M., Karastoyanova, D., Leymann, F., & Vukojevic-Haupt, K. (2014). Service Composition for REST. *2014 IEEE 18th International Enterprise Distributed Object Computing Conference*, 110–119. https://doi.org/10.1109/edoc.2014.24

Jouault, F., Allilaire, F., Bézivin, J., & Kurtev, I. (2008). ATL: A model transformation tool. *Science of Computer Programming*, *72*(1–2), 31–39. https://doi.org/10.1016/j.scico.2007.08.002

Kleppe, A. G., Warmer, J. B., & Bast, W. (2003). *MDA Explained: The Model Driven Architecture : Practice and Promise*.

Kouamou, G. E., & Kungne, W. K. (2017). A Structural and Generative Approach to Multilayered Software Architectures. *Journal of Software Engineering and Applications*, *10*(08), 677–692. https://doi.org/10.4236/jsea.2017.108037

Lämmel, R., Visser, J., & Saraiva, J. (2008). *Generative and Transformational Techniques in Software Engineering II: International Summer School, GTTSE 2007, Braga, Portugal, July 2-7. 2007, Revised Papers*. https://doi.org/10.1007/978-3-540-88643-3

Mazzara, M., Dragoni, N., Bucchiarone, A., Giaretta, A., Larsen, S. T., & Dustdar, S. (2021). Microservices: Migration of a Mission Critical System. *IEEE Transactions on Services Computing*, *14*(5), 1464–1477. https://doi.org/10.1109/tsc.2018.2889087

Mernik, M., Heering, J., & Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM Computing Surveys*, *37*(4), 316–344. https://doi.org/10.1145/1118890.1118892

Montesi, F., Guidi, C., & Zavattaro, G. (2014). Service-Oriented Programming with Jolie. *Web Services Foundations*, 81–107. https://doi.org/10.1007/978-1-4614-7518-7_4

Montesi, F., Peressotti, M., & Picotti, V. (2021). Sliceable Monolith: Monolith First, Microservices Later. *2021 IEEE International Conference on Services Computing (SCC)*, 364–366. https://doi.org/10.1109/scc53864.2021.00050

Nadareishvili, I., Mitra, R., McLarty, M., & Amundsen, M. (2016). *Microservice Architecture: Aligning Principles, Practices*, and *Culture*.

Neuman, S. (2015). *Building microservices: Designing fine-grained systems*.

Pahl, C., & Jamshidi, P. (2016). Microservices: A Systematic Mapping Study. *In Proceedings of the 6th International Conference on Cloud Computing and Services Science (CLOSER 2016)*, 137–146.

Raible, M. (2016). The JHipster mini-book. *Lulu. Com*. https://www.jhipster.tech/

Shadija, D., Rezai, M., & Hill, R. (2017). Towards an understanding of microservices. *2017 23rd International Conference on Automation and Computing (ICAC)*, 1–6. https://doi.org/10.23919/iconac.2017.8082018

Sorgalla, J., Wizenty, P., Rademacher, F., Sachweh, S., & Zündorf, A. (2018). AjiL: enabling model-driven microservice development. *ECSA '18: Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*, 1–4. https://doi.org/10.1145/3241403.3241406

Suljkanović, A., Milosavljević, B., Inđić, V., & Dejanović, I. (2022). Developing Microservice-Based Applications Using the Silvera Domain-Specific Language. *Applied Sciences*, *12*(13), 6679. https://doi.org/10.3390/app12136679

Terzić, B., Dimitrieski, V., Kordić, S., Milosavljević, G., & Luković, I. (2018). Development and evaluation of MicroBuilder: a Model-Driven tool for the specification of REST Microservice Software Architectures. *Enterprise Information Systems*, *12*(8–9), 1034–1057. https://doi.org/10.1080/17517575.2018.1460766

Thönes, J. (2015). Microservices. *IEEE Software*, *32*(1), 116–116. https://doi.org/10.1109/ms.2015.11

Whittle, J., Hutchinson, J., & Rouncefield, M. (2014). The State of Practice in Model-Driven Engineering. *IEEE Software*, *31*(3), 79–85. https://doi.org/10.1109/ms.2013.65

Wizenty, P., Sorgalla, J., Rademacher, F., & Sachweh, S. (2017). MAGMA: build a management-based generation of microservice infrastructures. *Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings*, 61–65. https://doi.org/10.1145/3129790.3129821

Zimmermann, O., Stocker, M., Lubke, D., Zdun, U., & Pautasso, C. (2022). *Patterns for API design: simplifying integration with loosely coupled message exchanges*.