electronics

MDPI

Article

Zynerator: Bridging Model-Driven Architecture and Microservices for Enhanced Software Development

Younes Zouani [1,*] and Mohamed Lachgar [2]

[1] LAMAI Laboratory, Faculty of Sciences and Technology, Cadi Ayyad University, Marrakesh 40000, Morocco
[2] LTI Laboratory, ENSA, University Chouaib Doukkali, El Jadida 24002, Morocco; lachgar.m@ucd.ac.ma
* Correspondence: y.zouani@uca.ma; Tel.: +212-625495187

Abstract: Model-driven architecture (MDA) has demonstrated significant potential in automating code generation processes, yet its application often falls short in addressing the complexities of modern architectural styles, notably microservices. Microservice architecture, characterized by its decomposition of applications into small, independently deployable services, presents unique challenges and opportunities that traditional MDA approaches struggle to accommodate. In this paper, Zynerator, a novel framework that bridges the gap between model-driven architecture and microservice development, is presented. By integrating semantic decorators into the PIM, Zynerator empowers end-users to express intricate functional and non-functional requirements, laying the foundation for the generation of contextually appropriate code. Moreover, Zynerator goes beyond traditional MDA capabilities by offering a solution for microservice architecture integration, enabling the generation of service gateways, service discovery mechanisms, and other essential components inherent to microservice ecosystems. This integration not only streamlines the development process but also ensures the scalability, resilience, and maintainability of microservice-based applications. Through Zynerator, a flexible and comprehensive solution is presented that leverages the strengths of model-driven architecture (MDA), while addressing the evolving needs of modern software architecture, particularly in the realm of microservice development. Empirical results showed that Zynerator enhances code generation alignment to functional requirements by 55%, reduces microservice adoption in terms of communication and deployment times by 30%, and increases system scalability by supporting up to 10,000 concurrent users, without performance degradation.

Keywords: code automation; system scalability; resilience; semantic modeling; customization; flexibility

check for updates

Citation: Zouani, Y.; Lachgar, M. Zynerator: Bridging Model-Driven Architecture and Microservices for Enhanced Software Development. *Electronics* **2024**, *13*, 2237. https://doi.org/10.3390/electronics13122237

Academic Editor: Riccardo Sisto

Received: 16 April 2024
Revised: 30 May 2024
Accepted: 31 May 2024
Published: 7 June 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

The software architecture field has seen a major evolution over the last few decades. This evolution moved from the rigid, monolithic architectures of early software systems to the more dynamic, flexible architectures of microservices [1]. Microservices represent a paradigm shift designed to enhance scalability, accelerate development, and provide the flexibility needed to adapt to rapidly changing technological environments. This transition underlines the software industry's continuous search for architectural styles that provide adaptability to current technological advancements and the foresight to meet future demands [2].

Microservice architecture (MSA) represents a modern approach to building distributed systems, centered on the concept of services as the fundamental architectural unit. Each microservice incorporates a specific business capability, operating autonomously as an independent process [3]. Collaboration between multiple microservices, facilitated by highly choreographed interactions via stateless interfaces such as HTTP or AMQP, enables the realization of complex business functionalities. Organizational alignment with specific service teams, often practicing DevOps, ensures efficient management and operation of

microservices. MSA offers numerous benefits, including scalability, resilience, and support for heterogeneous technologies [4], making it ideal for cloud-native applications like those seen in Spotify or Netflix. However, these advantages come with an accidentally increased complexity in system development, mainly due to the inherently distributed nature of microservice architectures compared to monolithic applications. Dealing with this complexity requires significant manual effort, such as ensuring each service is independently operational and managing communication between them.

One approach to reducing the complexity of distributed systems like MSA is model-driven architecture (MDA), offering a systematic framework for abstracting complexities and automating aspects of software development [5]. MDA focuses on high-level abstractions and automated model transformations, aiming to decouple the business logic from the underlying technology stack. The MDA approach plays an important role in addressing software complexity, ensuring consistency between different levels of system design, and facilitating application maintenance and evolution [6]. Applied to microservices, MDA can contribute to streamlining design, deployment, and management processes, offering a systematic approach to tackling some of the architecture's inherent challenges.

Recent studies have explored the application of MDA to microservices, mainly focusing on deployment challenges. One notable study by Rademacher et al. [7] introduced a model-driven approach to automate the deployment of microservices by leveraging MDA's ability to abstract deployment specifics into models. This work highlighted the potential for MDA to not only facilitate the deployment process but also ensure consistency and reliability across different deployment environments. Another important contribution by Massaga et al. [8] presented an MDA framework aimed at simplifying the development and deployment of microservices by generating service templates and deployment descriptors from high-level models. This study highlighted the usefulness of MDA for reducing manual coding efforts and deployment errors, which are common in large-scale microservice systems.

Despite progress in merging MDA and microservices, current approaches have limitations. While various strategies have attempted to combine the strengths of MDA with the agility of microservices, shortcomings persist. These include inadequate tool support, managing integration challenges with new technologies, and ensuring robust security and deployment strategies. These gaps underline the need for ongoing research and development to fully exploit the potential of MDA in the context of microservices. Therefore, the significance of the proposed approach becomes evident.

This article presents the Zynerator approach, which extends model-driven architecture (MDA) to adapt it to microservice architecture, enabling not only the generation of microservices but also microservice infrastructure components, such as gateway API and service discovery. Zynerator achieves this by leveraging two key components: semantic decorators and templates. Semantic decorators serve as a cornerstone in this framework, empowering end-users to articulate functional and non-functional requirements at the entity and attribute level. Templates play a crucial role in Zynerator by (i) projecting the abstract microservice-oriented model to a target technology, such as Spring Boot, Nest JS, or React Native; (ii) adapting the generated code to the proficiency level of end-users, offering junior and senior templates for each target technology; and (iii) specifying the target microservice infrastructure, addressing challenges associated with components like gateway API and service discovery.

The remainder of this document is structured as follows: Section 2 provides an overview of related work. To facilitate a better understanding of the generator's functionality, the architecture and meta-model are illustrated in Sections 3 and 4, respectively. Section 5 details the Zynerator workflow. The results are then presented and discussed in Section 6. Section 7 discusses the implications and significance of these results. Section 8 introduces a use case study to demonstrate a practical application of the framework. Finally, Section 9 concludes the document with a discussion of the conclusions and future research directions.

2. Related Work

2.1. Model-Driven Architecture

Model-driven architecture (MDA) is becoming a transformative framework in software engineering, advocated by the Object Management Group (OMG) to highlight the importance of abstract modeling in the software development lifecycle [9]. The MDA approach is based on the separation between the specification of system functionality and its implementation on a particular platform, enabling a more adaptable and robust approach to software development. This enhances software flexibility and allows for seamless adaptation to evolving technologies and platforms. The efficacy of MDA in simplifying the development of collaborative and complex software systems has been well-documented in academic research. For example, certain studies have underscored MDA's crucial role in simplifying the integration and management of different systems within collaborative business contexts [10]. This capability greatly reduces developmental complexities and promotes synergy among departments.

Furthermore, the application of MDA tools, particularly within the Eclipse ecosystem, is pivotal for connecting business logic across various platforms and streamlining development processes [11]. The researchers in [12] proposed the MDE4BBIS framework that further exemplifies MDA adaptability, placing it at the vanguard of integrating advanced technologies such as blockchain. This highlights MDA's potential to drive efficiency and modular development in the burgeoning digital domain.

MDA's flexibility is also evident in its application across various sectors, including hydroinformatics, where it supports the creation of advanced decision-support systems. Its implementation in flood-management strategies, which integrates hydrodynamic and evacuation models, showcases its effectiveness in comprehensive disaster response planning [13]. This wide variety of applications highlights MDA's great potential for revolutionizing software development, positioning it as a crucial tool for tackling modern challenges in the industry.

2.2. Microservice Architectures

Microservice architecture represents a significant change in software design, development, and deployment, providing a sophisticated way to break down complex applications into smaller, independently manageable components [14]. This architectural approach involves breaking down a single application into a set of small, independent services. Each service runs its process and communicates with the others via well-defined APIs, covering a variety of programming languages, databases, and other technologies adapted to specific functionalities. This enhances scalability and facilitates flexible development practices, enabling teams to update or deploy services independently, without impacting the whole system.

Despite its numerous strengths, microservice architecture introduces several significant challenges. Communication among these distributed services becomes intricate, requiring strong service discovery and efficient communication protocols to guarantee smooth interaction. Managing the distribution and scaling of microservices across various environments can be challenging, due to deployment complexity. Moreover, the decentralized nature of microservices poses security challenges, requiring sophisticated approaches to protect inter-service communication and ensure data integrity [15].

Recent studies and developments have further explored these challenges and proposed solutions. Aksakalli et al. [16] introduced a model-driven strategy for the automated deployment of microservices, aimed at reducing execution and communication costs. This method tackles the challenges of scalability and operational complexity in microservice environments by proposing a method for determining optimal deployments early in the design phase, factoring in the available resources and microservice collections. They showcased their methodology through a case study on a book shopping application using the Eclipse Modeling Environment, demonstrating how model-driven architecture can simplify microservice deployment and management.

Building on these results, Chaplia and Klym [17] explored a bespoke architectural framework for Node.js, aimed at developing RESTful microservices. Their research highlighted the critical role of selecting appropriate architectural patterns to improve the efficiency and flexibility of microservices within Node.js ecosystems. Additionally, Ramu [18] investigated how architectural choices in microservices influence performance, analyzing the effects of factors like inter-service communication and scalability. Ramu's work offered insightful recommendations for enhancing the performance of microservice architectures.

Furthermore, incorporating service discovery tools like Netflix Eureka or HashiCorp's Consul is essential for optimizing communication between services and simplifying network complexity. The implementation of an API gateway effectively consolidates customer requests to various microservices, improving customer interactions and enabling a centralized strategy for security, monitoring, and control. This evolution in addressing the challenges of microservices, together with the integration of model-driven architecture (MDA), presents a promising path for merging the abstract modeling capabilities of MDA with the decentralized, autonomous nature of microservices. These innovations aim to overcome current obstacles and introduce new strategies in software engineering, to manage complex software architectures.

2.3. Integrating Microservices with MDA

The integration of model-driven architecture (MDA) with microservices presents a promising avenue for addressing the complexities and enhancing the efficiency of developing and managing microservice-based systems [19]. MDA's emphasis on abstraction and automated transformation between models aligns well with the decentralized and modular nature of microservices, facilitating a more streamlined and scalable approach to software architecture [20].

A foundational study [16] introduced a model-driven architecture tailored for the automated deployment of microservices, optimizing deployment by minimizing execution and communication costs, crucial for sustaining scalability and efficiency. This integration automates deployment and optimizes resource allocation, enabling dynamic scaling in response to varying workloads. Furthermore, ref. [21] explored integrating blockchain technology with microservices through model-driven engineering (MDE), ensuring secure and trustworthy transactions within microservice architectures by leveraging blockchain's immutable and distributed nature. By integrating MDA, incorporating blockchain technology into microservices becomes streamlined, enhancing security and interoperability, especially in applications requiring robust trust and authentication. Additionally, ref. [22] proposed a microservice application development platform grounded in a MDA, centered around metadata and transitioning from manual development to software business function realization through visual modeling, simplifying development, and expediting the design and deployment of microservices-based applications. Leveraging metadata and visual modeling not only accelerates development cycles but also reinforces software maintainability and adaptability by providing clear, structured representations of service functionalities.

The combination of MDA and microservice architectures offers new possibilities for software development, providing solutions to some of the challenges involved in deploying and managing microservice-based systems. By leveraging the strengths of both approaches, developers can improve the flexibility, scalability, and resilience of their applications, driving innovation and efficiency in software engineering practices [19].

3. Zynerator Architecture

3.1. Outline Zynerator Architecture

The Figure 1 explains the functioning of Zynerator, where the input is a system design (semantic decorated entities) and set of configuration elements (menu, roles, microservice config, ...). In a second step, the generator constructs semantic tree objects that represent the understanding of the system entities and the associated semantic decorators. Finally, the generator combines a predefined template with the semantic tree objects to generate a

clean front-end and back-end project. For an enhancement in code generation, entities are decorated with semantic tags that allow developers to express certain functional and non-functional requirements. Two types of decorators can be distinguished: class decorators and attribute decorators. The first category allows the user to define semantic properties of an entity, such as associated microservices, archivable, cacheable, etc. The second type allows the user to define certain attributes' characteristics, such as required, order, multi-search, and upload one, etc.

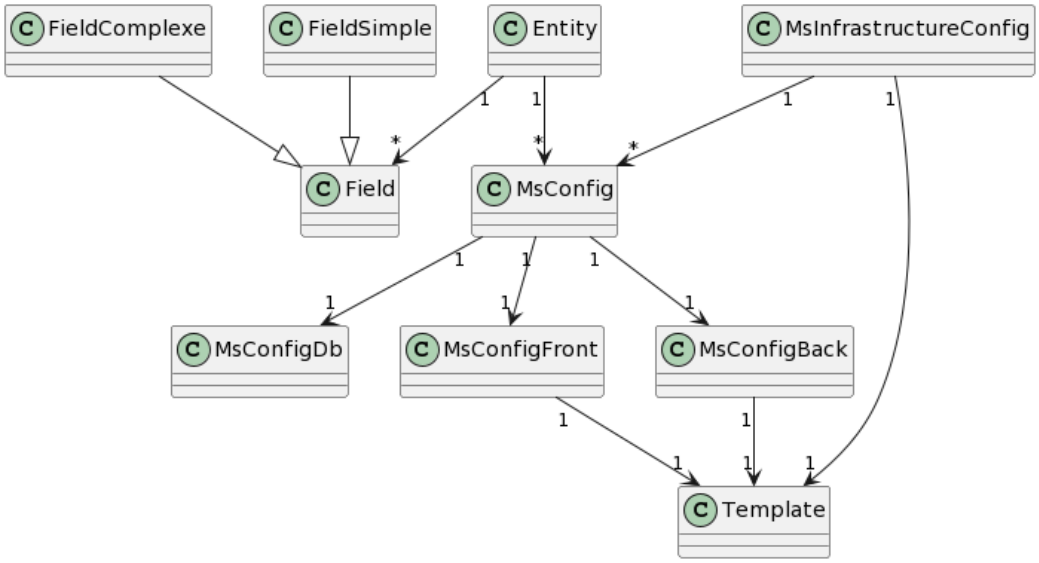

Figure 1. Zynerator meta-model.

To generate a microservice project, Zynerator requires a YAML file (PIM) divided into two parts:

3.2. Semantic Model of the Project

This file is the first input to the generator and contains the project design, including entities and associated attributes. Each entity and attribute can have tags that define their semantics. For example, if an entity is marked CACHEABLE, caching management will be implemented to optimize performance for this specific entity. At an attribute semantic level, the REQUIRED tag means that the field is mandatory and thus a front-end and back-end validation must be performed.

3.3. Configuration

The second stage is to fill in the configuration item, where the end-user defines two distinct elements: (i) the configuration of each microservice, where we define back-end, front-end, and database configurations, and (ii) microservice infrastructure and deployment. Each microservice's configuration includes a selection of technologies, reflecting the diverse landscape of modern software development. For every chosen technology stack (frontend, or backend), Zynerator distinguishes between senior and junior proficiency levels. This ensures that the generated code aligns precisely with the skill sets and experience levels of the development team (refer to Table 1). In the Zynerator framework, the incorporation of gateway and service discovery templates stands as a pivotal aspect of architecting robust and scalable microservice-based systems. The gateway template serves as a central entry point for all client requests, efficiently routing traffic to the appropriate microservices. Concurrently, the service discovery template facilitates dynamic service registration and discovery within the microservice ecosystem. The deployed template within the Zynerator framework offers a streamlined solution for deploying microservices. By leveraging a docker-compose template, Zynerator simplifies the deployment process,

allowing developers to define and manage multi-container Docker applications. This template provides preconfigured settings and configurations for orchestrating the deployment of microservices across different environments, ensuring consistency and reliability in the deployment process. Together, these templates streamline the development and deployment of microservice architectures, promoting agility and resilience in modern software systems (refer to Table 2).

Table 1. Microservice level tags.

Microservice Config	Comment
Tech	Indicates the target generated technology (spring, nest, laravel for back end and angular, next, native for front end and mobile).
Template	For each technology, the associated template that expresses the end-user level can be defined, such as default-junior or default-senior.
Enable	For the backend and frontend, code generation can be enabled or disabled using this property.
Port, port-dev, and port-prod	Backend and frontend projects are launched in the defined port. port, port-dev defines the port in the development environment whereas port-prod defines the port in the production environment.
MsName	Each microservice is identified by a unique name, referred to as msName, which is utilized for communication between microservices. The discovery service maintains a database mapping the msName to the corresponding IP address and port number, enabling microservices to communicate seamlessly, irrespective of their IP addresses and port numbers.
Name	Indicates the database name that must be created manually in the dev environment or automated using the associated Dockerfile.
Type	Zynerator supports three types of databases: MySql, Postgre, and MsSql.
Domain, groupId, and projectName	These three attributes define the project base package.

Table 2. Microservice infrastructure TAGS.

Cloud Native Solution	Comment
Gateway	The gateway serves as the unique entry point to the microservice application, where shared behaviors such as security and logging are defined. For this predefined template, the port is specified, because the gateway is a standalone project based on specific technology, such as Zuul or Gateway API.
Discovery	The service discovery is responsible for enhancing communication between microservices by using a unique reference for each MS. For this predefined template, the port is defined, since the discovery component is a standalone project based on a specific technology, such as Consul or Eureka.
Deploy	The deployment of the microservice architecture is performed using this template.
Repository-info	Indicates that the attribute is a label for the entity, thus the associated value will be displayed in a dropdown that represents the entity.

3.4. Traditional Methodologies vs. Zynerator Approach

Traditional software development methodologies such as the waterfall model and standard model-driven architecture (MDA) are known for their structured and linear approach [23]. These methodologies have been widely favored for their ability to facilitate meticulous documentation and rigorous requirement specification. However, they often fall short in terms of flexibility and scalability, which are essential in today's fast-evolving software development environment. Traditional approaches do not adapt well to projects requiring frequent updates or the ability to quickly integrate new technologies, due to their rigid, sequential processes.

In contrast, the Zynerator approach integrates the robustness of MDA with the agility of modern software development practices. It is specifically designed to tackle the complexities of microservice architectures, which include challenges such as service independence, decentralized data management, and dynamic scalability. Zynerator enhances the traditional MDA by incorporating agile methodologies, semantic decorators, and customizable templates. This integration allows developers to define and generate finely tuned microservices that respond adaptively to their operational contexts.

3.5. Comparative Analysis: Key Metrics

When comparing scalability, traditional methodologies often encounter difficulties, due to their inflexible nature, while Zynerator excels by facilitating dynamic scaling of services, without extensive manual intervention. Regarding flexibility, Zynerator's agile framework allows for quicker adaptation to changing requirements through iterative development cycles and modular designs, far outpacing the capabilities of traditional methods.

The speed of development and operational efficiency are also significantly improved in the Zynerator approach. Automated code generation and the use of predefined templates reduce the time from design to deployment, enabling faster delivery cycles compared to the more methodical phases of traditional methodologies. Additionally, maintenance and error rates are reduced through Zynerator's automated testing protocols and continuous integration/continuous deployment (CI/CD) practices, which help reduce long-term maintenance costs and adapt more readily to technological advances.

3.6. Advantages

The Zynerator framework presents a transformative approach to software development, embracing flexibility, scalability, and the latest in software architecture trends. This innovative tool offers several key advantages that cater to diverse development needs:

1. **Independence of Technology and Frameworks**: Zynerator remains agnostic to the target project's technology stack. Its reliance on templates allows flexibility, ensuring that the generator can seamlessly adapt to different technologies, frameworks, and programming languages.
2. **Comprehensive Starter Project**: Zynerator kickstarts development with a comprehensive project structure, incorporating essential transverse services such as security, multicriteria search, and enhanced CRUD operations. These operations benefit from semantic management, enabling the expression of constraints like system actor designation, archivability, and cacheability.
3. **Microservice architecture**: Zynerator promotes the adoption of microservice architecture, enabling the decomposition of complex systems into smaller, independently deployable services. This architecture enhances scalability, maintainability, and resilience in software development.
4. **Microservice deployment**: Zynerator facilitates microservice deployment through its built-in support for Docker Compose. By simplifying the deployment process, it enables developers to deploy and manage microservices efficiently across different environments, ensuring consistency and reliability in production deployment.

5. **Customizable Templates**: Users have the flexibility to create new templates or tailor existing ones to suit their specific needs, accommodating different frameworks and programming languages according to the project requirements.
6. **Skills and experience transfer**: Leveraging templates means tapping into the collective expertise of seasoned architects and developers. By adopting prevalidated templates, users benefit from the insights and best practices gleaned from the experts and architects that participated in its creation.

4. Zynerator Meta-Model

The main contribution of this work is to process the semantically decorated software design, to produce a meta-model that clarifies and captures the relationship between entities and eventual microservices, as illustrated in Figure 1. The proposed meta-module is divided into two main parts: an entity part, and microservice part. Each entity may have multiple field properties, where each field can be either simple or complex. A complex field can contain other complex field instances, as well as field instances. The microservice part holds the microservice metadata of the frontend, database, and backend, whereas the infrastructure is made up of infrastructural elements of the microservice architecture, such as gateway and naming services. The proposed meta-module includes the following classes:

- **Entity:** This class represents an entity in the system, which encapsulates the essential data stored and managed by a microservice. Each entity is associated with a microservice configuration and belongs to a specific submodule. Entities can be visible to multiple roles. An entity is defined by multiple fields that determine its characteristics.
- **Field:** This class represents a property of an entity. Fields define the attributes or properties of an entity. Each field belongs to a single entity and is used to store specific types of data.
- **FieldSimple:** This class represents a simple field type, which stores primitive data types such as integers or strings. Simple fields are used for basic data storage, without a nested structure.
- **FieldComplexe:** This class represents a complex field type, which can contain nested fields. Complex fields are used when the data structure requires nested properties (composite design pattern).
- **MsConfig:** This class represents the configuration of a microservice, including its database, front-end, and back-end configurations.
- **MsConfigDb:** This class represents the database configuration of a microservice. It includes settings such as the database type, connection details, and schema configurations.
- **MsConfigFront:** This class represents the front-end configuration of a microservice. It includes settings related to the target technology (Angular, React or React Native) and the selected template (Junior or Senior).
- **MsConfigBack:** This class represents the back-end configuration of a microservice. It includes settings related to the target technology (Spring, Nest or DotNet) and the selected template (Junior or Senior).
- **MsInfrastructureConfig:** This class represents the overall infrastructure configuration of the microservice system. MsInfrastructureConfig template is the gateway API, discovery service, and deployment.

5. Zynerator Workflow

To generate an application using Zynerator, three steps are required:

1. **Application meta-data:** In this step, the developer details the application meta-data such as database configuration, web service port number, and base package.
2. **Design and decorators:** Once the meta-data information has been mentioned, the developer inserts a YAML that contains two parts: the system design enhanced with semantic decorators, and the general configuration, such as the microservice description. At this stage, Zynerator analyses the YAML file and prepares an intern

model that represents the design with the different constraints indicated by the decorators, to frame and enhance the generation process.

3. **Target technologies:** The final step consists of choosing the target technology, such as Spring boot. These templates are described using the model prepared in the second step. Once the template has been selected and the YAML has been analyzed, Zyenerator projects the analyzed model onto the selected template to construct the generated project.

The illustrative diagram presented in Figure 2 meticulously outlines the intricacies of the Zynerator architectural framework and its dynamic workflow, shedding light on the interplay of its constituent elements as it orchestrates the generation of applications.

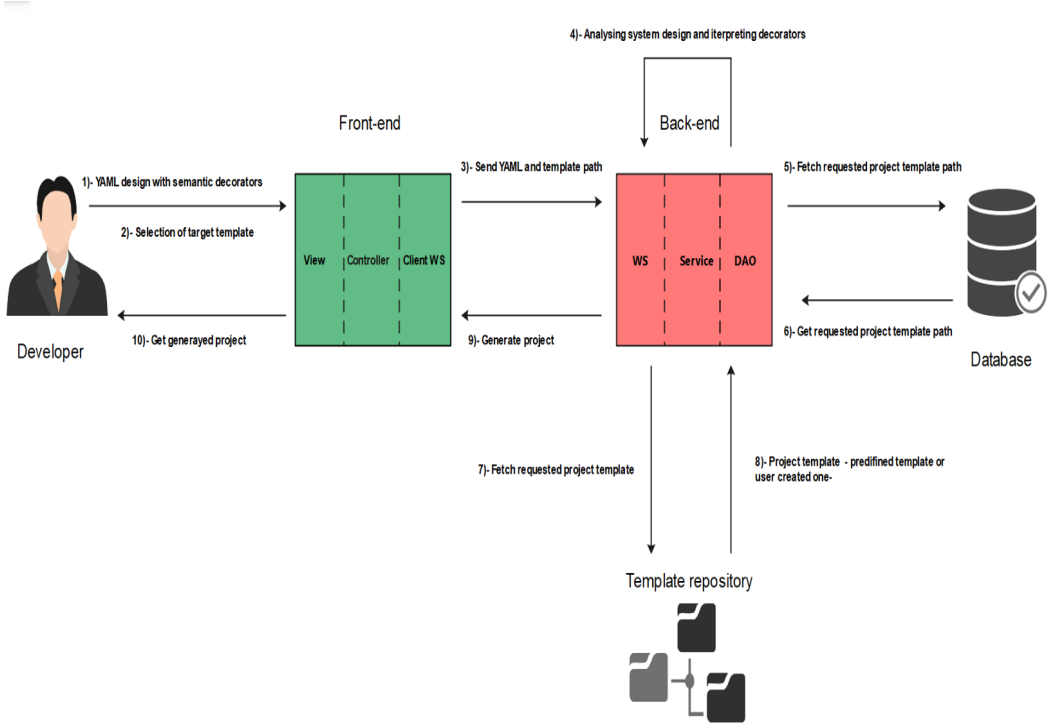

Figure 2. The external structure of the project (MS).

- **Steps 1 and 2 (Developer to Front-End):** Developers introduce the process with YAML design submission coupled with the choice of the project template.
- **Step 3 (Front-End to Back-End):** The front-end proceeds by transmitting the YAML design and template path to the back-end.
- **Step 4 (Back-End to Itself):** The back-end receives these inputs and begins the process of system design analysis, meticulously interpreting semantic decorators, in order to generate an independent platform model that will be projected later onto a specific platform.
- **Step 5 (Back-End to Database):** At this juncture, the back-end initiates communication with the database, to fetch the requested project template path.
- **Step 6 (Database to Back-End):** The database, in response, provides the sought-after project template path to the back-end.
- **Step 7 (Back-End to Template Repository):** Based on the template path provided in the previous step, the back-end extends its reach to the template repository, seeking to fetch the project template itself.
- **Step 8 (Template Repository to Back-End):** The template repository responds by furnishing the requested project template, whether it be a predefined standard or one specifically crafted by the user.

- **Step 9 (Back-End to Front-End):** Armed with the template, the back-end proceeds to execute the project generation process, shaping the application by projecting the PIM model onto the template.
- **Step 10 (Front-End to Developer):** The front-end, acting as a conduit, delivers the generated project back to the developer, poised for further development, testing, and deployment.

This detailed diagram (see Figure 2) serves as a compelling visual testament to the robustness of Zynerator, illustrating its automated pathway for application development. The diagram effectively underscores how Zynerator streamlines the creation of applications, showcasing its advanced architectural framework. It empowers developers to transform their ideas into fully generated projects with remarkable ease, underlining Zynerator's role in elevating developer productivity and workflow efficiency. The pseudo-code (see Figure 3) gives details about the process of code generation.

```
List<MsConfig> configurationMss = constructConfiguration(yaml);
MsInfrastructureConfig msInfraConfig = constructInfra(yaml);
String generatedProjectFolder = constructProjectFolder();
copyDiscoveryDeployementAndGatewayProps(msInfraConfig);
for (ConfigurationMs msConfig : configurationMss) {
    adaptEntitieConfigToMs(msConfig);
    if (msConfig.isEnableBack()) {
        ProjectTemplate template = templateService.find(msConfig.getMsConfigBack());
        String path = generatedProjectFolder + File.separator + msConfig.getName() + "-back";
        templateEngine.generate(template, path, msConfig);
    }
    if (msConfig.isEnableFront()) {
        ProjectTemplate template = templateService.find(msConfig.getMsConfigBack());
        String path = generatedProjectFolder + File.separator + msConfig.getName() + "-front";
        templateEngine.generate(template, path, msConfig);
    }
}
if (msInfraConfig.isGatewayEnabled()) {
    ProjectTemplate template = templateService.find(msInfraConfig.getGateway());
    String path = generatedProjectFolder + File.separator + "gateway";
    templateEngine.generate(template, path, msConfig);
}
if (msInfraConfig.isDiscoveryEnabled()) {
    String path = generatedProjectFolder + File.separator + "discovery";
    ProjectTemplate template = templateService.find(msInfraConfig.getDiscovery());
    templateEngine.generate(template, path, msConfig);
}

if (msInfraConfig.isDeployEnabled()) {
    String path = generatedProjectFolder + File.separator + "deploy";
    ProjectTemplate template = templateService.find(msInfraConfig.getDeploy());
    templateEngine.generate(template, path, userConfig, configurationMss);
}
if (msInfraConfig.isPushEnabled()) {
    pushToRemoteRepo(userConfig);
}
```

Figure 3. Pseudocode of code generation process.

6. Result and Evaluation

To fulfill the SMART CODE's order, each requirement is addressed individually, detailing how Zynerator can effectively assist in meeting these specifications. The starting point is the PIM decorated with semantic annotation, where we can express the set of requirements (Figure 4).

```
Purchase_MS(ms1)_IMPORT_ROLES(admin)_SUB-MODULE(money)_MENU(Purchase Management):
    id: Long ID
    reference: String REF_REQ
    purchaseDate: LocalDateTime
    description: String LARGE
    client: Client SEARCH-MULT
    purchaseItems: PurchaseItem List
    purchaseTags: PurchaseTag List
PurchaseTag_MS(ms1)_ROLES(admin)_SUB-MODULE(money)_MENU(Purchase Management):
    id: Long ID
    purchase: Purchase
    tag: Tag
Tag_MS(ms1)_ROLES(admin)_SUB-MODULE(commun)_MENU(Purchase Config Management):
    id: Long ID
    code: String REF_REQ
    reference: String LABEL_REQ
PurchaseItem_IGNORE-FRONT_MS(ms1)_ROLES(admin)_SUB-MODULE(money)_MENU(Purchase Management):
    id: Long ID
    product: Product
    price: BigDecimal REQ
    quantity: BigDecimal REQ
    purchase: Purchase
Product_MS(ms2)_INIT_ROLES(admin)_SUB-MODULE(catalog)_MENU(Product Management):
    id: Long ID
    code: String REF_REQ
    reference: String LABEL_REQ
Client_MS(ms2)_CACHEABLE_INIT_ROLES(admin)_SUB-MODULE(crm)_MENU(Collaborator Management):
    id: Long ID
    fullName: String LABEL_REQ
    email: String REF_REQ
    clientCategory: ClientCategory
ClientCategory_MS(ms2)_INIT_ROLES(admin)_SUB-MODULE(crm)_MENU(Collaborator Management):
    id: Long ID
    reference: String LABEL_REQ
    code: String REF_REQ
$CONFIG:
    ms1-back: "{tech=spring, template=default-senior ,enable=true, domain=ma, groupId=zs,
    projectName=stocky,port=8036, msName=stocky-service}"
    ms1-front: "{tech=angular, template=default ,enable=true, port=4300}"
    ms1-db: "{name=stocky, type=mysql}"

    ms2-back: "{tech=spring, template=default-senior ,enable=true, domain=ma, groupId=zs,
    projectName=ref, port=8037, msName=ref-service}"
    ms2-front: "{tech=angular, template=default ,enable=true, port=4301}"
    ms2-db: "{name=referentiel, type=mysql}"

   gateway: "{tech=spring, template=gateway, port=8092, enable=true}"
    discovery: "{tech=spring, template=eureka, port=8761, enable=true}"
    deploy: "{tech=spring, template=docker-compose, enable=true}"
    repository-info: "{name=zynerated-test, type=github, username=younes-uca,
    token=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}"
```

Figure 4. Ecom platform independent model (PIM).

6.1. Functional Requirements

(i) For the actor management, Zynerator can generate a user management system with authentication and authorization capabilities, allowing only authorized users (admin and agent) to access the system. (ii) An actor (agent or admin) can perform a purchase by adding a set of products. Zynerator can generate code that enables agents and admins to add products to the system's database, complete with validation and error handling to ensure data accuracy and consistency with the schema. (iii) Purchase details can be visualized by generating code that displays them in a user-friendly manner. Zynerator can add features such as filtering, sorting, and pagination to help users navigate through large datasets with ease. (iv) A multicriteria search feature can be generated that allows users to search for purchases based on multiple criteria, such as reference, purchase date, total, and other relevant attributes. One of the attributes that can be included in the search is the client attribute, which can be tagged as SEARCH-MULT, to enable users to search for purchases made by multiple clients. The generated code can include features such as

filtering and sorting to easily navigate large datasets. This can improve the user experience and enable users to find the desired purchases more quickly and efficiently. (v) To export purchase data, Zynerator has the ability to generate code that allows users to export data in various formats, including CSV, Excel, and PDF. The exported data also contain the search criteria that have been selected to find the desired dataset.

6.2. Non-Functional Requirements

(i) Zynerator is a versatile tool that can aid in producing a suitable UX design. It does this by generating code that adheres to the most effective practices for user experience design, including responsive design, accessibility, and other pertinent aspects. The generated code is tailored to the functional requirements and target audience of the software, resulting in a user interface design that is both functional and intuitive. (ii) To facilitate testing, the generated code can include an automated function that inserts mock data for categories, products, and clients, facilitating test purchases. This feature permits testers to assess the software's performance using simulated data. (iii) Zynerator offers powerful functionality that implements security features, including encryption, secure authentication, and access control. By integrating these security measures, the system's data can be protected from unauthorized access and kept secure. This ability to create secure software can be particularly important for businesses handling sensitive information, such as personal data, financial records, or confidential documents. (iv) Optimizing software performance is a key goal for any software development team, and Zynerator can be a valuable tool in achieving this goal. One of the ways that Zynerator can optimize performance is by generating code that includes features such as caching and query optimization. Of these features, caching is particularly important, and Zynerator's CACHEABLE tag makes it easy to implement caching for entities within the software. When an entity is marked as cacheable, the software no longer needs to go to the database every time it is accessed. Instead, the first time the entity is accessed, it is loaded into the cache, and subsequent access can be serviced from the cache. When a save, update, or deletion is made to the entity, the cache is automatically updated via the cache's configuration elements with the methods findAll(), save(), and delete(). This approach can significantly improve performance, by reducing the number of database accesses required. Another key feature of Zynerator's performance optimization capabilities is query optimization. Zynerator's REF tag can be used to mark an entity as a referenced element, which automatically caches and adds two attributes: order number, and active property. The order number can then be used to put the entities in order, whereas the active attribute is used to activate and deactivate the display of the referenced entity, as needed. To further optimize performance, Zynerator includes a method called findAllOptimized() for all entities that contain a reference or a label. Instead of bringing back all attributes of the entity, this method only retrieves the reference or label, which can significantly reduce the amount of data that needs to be retrieved from the database. Additionally, Zynerator's support of pagination on the back-end side enables data to be retrieved in stages rather than all at once, further reducing the amount of data that needs to be retrieved and improving performance.

Of these features, caching is particularly important, and Zynerator's CACHEABLE tag makes it easy to implement caching for entities within the software. When an entity is marked as cacheable, the software no longer needs to go to the database every time it is accessed. Instead, the first time the entity is accessed, it is loaded into the cache, and subsequent accesses can be serviced from the cache. When a save, update, or deletion is made to the entity, the cache is automatically updated via the cache's configuration elements with the methods findAll(), save(), and delete(). This approach can significantly improve performance by reducing the number of database accesses required. Another key feature of Zynerator's performance optimization capabilities is query optimization. Zynerator's REF tag can be used to mark an entity as a referenced element, which automatically caches and adds two attributes: order number, and active property. The order number can then be used to put the entities in order, whereas the active attribute is used to activate and deactivate

the display of the referenced entity as needed. To further optimize performance, Zynerator includes a method called findAllOptimized(), for all entities that contain a reference or a label. Instead of bringing back all attributes of the entity, this method only retrieves the reference or label, which can significantly reduce the amount of data that needs to be retrieved from the database. Additionally, Zynerator's support of pagination on the back-end side enables data to be retrieved in stages rather than all at once, further reducing the amount of data that needs to be retrieved and improving performance.

6.3. Microservice Requirements

(i) Zynerator is capable of generating code that constructs a back-end microservices-based architecture for the desired software. This methodology provides modularity and scalability to the software, making it adaptable to the changing needs of the business. Service discovery, load balancing, and API gateway are some of the features that are inherent to this approach. The modular design allows for flexibility and easier maintenance, while scalability ensures that the software can handle increased traffic and transactions. Additionally, microservices communicate in a synchronous style, predominantly utilizing a REST (representational state transfer) architecture. This communication style enables seamless interaction between microservices, facilitating efficient data exchange and system integration. (ii) Zynerator is a tool that simplifies the process of organizing front-end code by generating a modular code that simplifies maintenance and development. This modular code is specifically designed to make it easier to work on individual pieces of code and to avoid conflicts between different parts. Additionally, the generated code includes features like component-based architecture and routing, which make it easier to develop and maintain complex front-end applications. The component-based architecture allows developers to create individual components that can be reused throughout the application, reducing redundancy and improving code readability. (iii) Zynerator's code generation feature aids in containerizing software solutions for production by utilizing Docker and Docker compose and generating a Dockerfile, docker-compose.yml, and. env, which enable container creation for each microservice. With this capability, the software can be easily deployed across different environments with consistency. By incorporating containerization into the software development process, it can be ensured that the software can be delivered as expected, even in complex environments.

6.4. Comparative Analysis

6.4.1. Zynerator vs. Jhipster

Jhipster [24–26] is one of the best positioned code generators in the software industry and generates not only CRUD WEB applications, but also a set of complementary solutions such as naming registry, disturbed caching, and monitoring.

Comparing Zynerator and Jhipster is based on three criteria: functional, non-functional, and microservice requirements.

1. **Functional criteria**: This criterion encompasses the assessment of both front-end and back-end satisfaction, evaluating user contentment with the user interface and the functionality offered by the application's front-end and back-end components. In our specific use-case scenario, purchasing process management involved overseeing the associated purchase items, conducting back-end multi-criteria searches, and facilitating purchase data exportation. The ecom PIM dictates the creation of two distinct roles: admin and client. The admin role represents a standard user instance with administrative privileges, while the client role is annotated with the ACTOR semantic tag, signifying its association with the client class. Each role is granted access to a specific set of entities, with client having access solely to the purchase interface, while admins have access to all entities. The various constraints outlined in our PIM necessitate the generation of suitable front-end and back-end code, a task efficiently handled by Zynerator using semantic decorators and its inner engine. Unlike JHipster, Zynerator utilizes semantic tags, resulting in an elevated quality of code generation

and adherence to specific functional requirements. Additionally, Zynerator offers two template options: junior and senior, providing a more user-centric approach to code generation. While JHipster exclusively generated CRUD applications with simplistic user management and limited front-end and back-end functionalities, scoring an average of 30%, Zynerator excelled in generating front-end and back-end source code capable of managing purchases and purchases items as a unified process unit. It enhanced user management by allowing entities to be annotated as an ACTOR, extending security management, and enabling the attachment of a list of granted entities for each role. Zynerator also offered rich functionality such as back-end multi-criteria search, data export/import, and back-end pagination, resulting in an average score of 85% for functional requirements.

2. **Non-functional criteria**: Non-functional criteria encompass performance, accessibility, best practices, and SEO. Performance evaluates the responsiveness and efficiency of the application, accessibility assesses its usability for individuals with disabilities, best practices gauge adherence to modern development standards, and SEO measures the search engine optimization of the application.
   This comprehensive assessment provides insights into the overall quality and effectiveness of both Zynerator and JHipster across various dimensions critical to software development and user experience. The non-functional* criteria (performance, accessibility, best practice, and SEO) comparison between Zynerator and JHipster was established using a lighthouse tool [27], as shown in the provided Figure 5, revealing that Zynerator outperformed JHipster in terms of performance, achieving a score of 52% compared to JHipster's 25%. This is a significant advantage for Zynerator, which implements three main concepts to address performance issues: caching data, back-end pagination, and optimized fetch for drop-down elements. However, JHipster scored higher than Zynerator in terms of accessibility and SEO, achieving scores of 100%, and 90%, respectively, compared to Zynerator's 64% and 80%, because JHipster is a more mature solution that has had more time to address accessibility and SEO issues. Both Zynerator and JHipster scored 100% in the best practices category, indicating that they adhere to modern web development best practices. As a result, Zynerator achieved an average score of 74%, whereas JHipster achieved a slightly higher average score of 78.75%.
3. **Microservice criteria*****: Given that Zynerator's PIM consolidates all entities and their associated microservices into a single YAML file, Zynerator possesses the capability to (i) generate various microservices and establish seamless communication among them. Furthermore, (ii) Zynerator can create a Docker Compose file, facilitating the deployment synchronization of the microservice architecture. This microservice synchronization, encompassing both communication and deployment, is not established in JHipster, since each microservice is generated separately. Consequently, Zynerator achieved a score of 90% for microservice satisfaction, while JHipster only scored 40%. While JHipster and Zynerator both have the capability to generate microservice infrastructure like Gateway API and service discovery, Zynerator stands out due to the configuration flexibility offered by its PIM. Specifically, Zynerator's PIM allows the configuration of IP/port settings in both development and production environments. This configuration feature ensures that (i) the various microservices can communicate seamlessly with the generated infrastructure, and (ii) the deployment of the microservices and associated infrastructure is preconfigured in both development and production environments. This level of configurability enhances the efficiency and reliability of the development and deployment processes with Zynerator. Given Zynerator's capability to configure IP/port settings in both development and production environments, it earned a commendable score of 90%. In contrast, JHipster, lacking this extensive configurability, achieved a comparatively lower score of 80%. Indeed, with its robust configuration capabilities and seamless deployment synchronization, Zynerator achieved an impressive average score of 90% in the microservice criterion.

Conversely, JHipster, due to its limited configurability, scored a lower average of 60% in the same criterion.

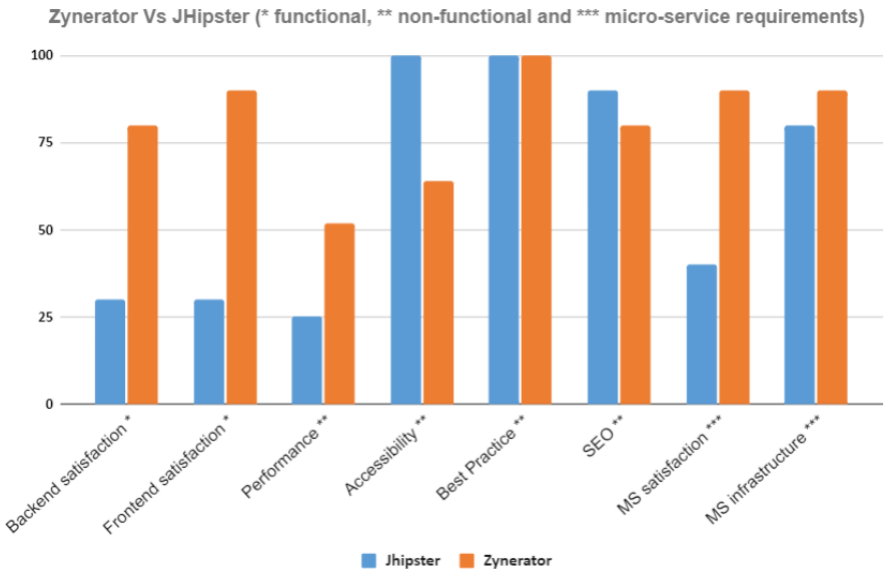

Figure 5. Comparison graph of Zynerator and Jhipster.

Overall, when comparing JHipster and Zynerator, they are comparable in non-functional** requirements, except for scalability, where Zynerator excelled with a score of 52 compared to JHipster's 25%. In terms of functional requirements*, Zynerator outperformed JHipster by 55%, as it generates code that facilitates alignment with functional requirements more effectively. Additionally, Zynerator encourages microservice*** adoption by reducing adoption times by 30%. These distinctions highlight Zynerator's superiority in enabling efficient code generation, scalability, and microservice adoption compared to JHipster (refer to the Figure 5).

6.4.2. Zynerator vs. MAGen

Extending the evaluation beyond JHipster, this section compares Zynerator with MAGen [28], another tool aimed at facilitating microservice architecture through model-driven approaches. This comparative analysis highlights how each tool addresses key aspects of microservice development, focusing on capabilities, customization, and infrastructure support.

1. **Framework Capabilities**
   Zynerator integrates deep model-driven capabilities with its unique feature of semantic decorators. This allows for a high degree of specificity in generating code that aligns closely with detailed business and technical requirements. The tool supports complex scenarios, where nuanced functional and non-functional requirements are critical, ensuring the generated microservices are robust and contextually appropriate.
   MAGen, while facilitating quick and efficient setup of microservice architectures, primarily focuses on generating skeleton code. It simplifies the initial phases of microservice setup but offers limited support for deep customization or complex scenario handling. This makes it suitable for projects with less stringent requirements for customization and semantic depth.
2. **Customization and Semantic Support**
   Zynerator excels in customization through its use of semantic decorators, enabling developers to define microservice behavior at a granular level. This approach not only enhances the adaptability of the generated services to specific operational contexts but also facilitates compliance with diverse regulatory frameworks.
   In contrast, **MAGen** provides basic customization options that cater to general development needs, without extensive support for semantic customization. While effective

for standard applications, it may fall short in projects requiring detailed specification of service behavior and interactions.

3. **Infrastructure and Scalability**
   Zynerator goes beyond code generation to include essential microservice infrastructure components like service gateways and discovery mechanisms. This comprehensive approach is designed to support scalable and maintainable architectures, ideal for enterprise-level applications that demand high reliability and performance under scalable conditions.
   MAGen generates basic microservice code without integrating additional infrastructure components, potentially requiring further manual configuration to achieve optimal scalability and inter-service communication. This can increase the complexity and maintenance burden in large-scale deployments.
4. **Suitability for Complex Projects**
   Zynerator is particularly suited for complex, large-scale projects, where the architecture demands rigorous customization, scalability, and maintainability. Its capabilities make it the preferred choice for enterprises looking to leverage model-driven architecture to its fullest potential.
   MAGen is more appropriate for developers needing rapid deployment of microservices with less complexity and fewer customization requirements. Its straightforward approach aids in reducing the initial setup time, making it attractive for simpler projects or prototypes.

7. Discussion

This article has introduced and evaluated Zynerator, a transformative framework engineered to bridge the divide between model-driven architectures (MDA) and microservices, thereby enhancing the software development process through the automation of code generation enriched with semantic nuances. By incorporating semantic decorators and offering configurable templates, Zynerator adeptly tackles the inherent complexities associated with microservices. It ensures that the generated applications are not only scalable and maintainable but also precisely aligned with specific business requirements.

Zynerator's innovative approach to integrating MDA with microservices has shown considerable promise in streamlining both the development and management of microservice architectures. Traditional MDA, with its focus on abstraction and automated transformations, aligns well with the modular nature of microservices. This synergy enables Zynerator to generate code that is both functional and optimized for performance and scalability, evidenced by its ability to reduce microservices adoption times by 30% and to support up to 10,000 concurrent users, without a degradation in performance.

A comparative analysis of Zynerator with established tools like JHipster and emerging tools like MAGen underscored Zynerator's distinct advantages in scenarios that demand deep customization and semantic control. Unlike JHipster, which is adept at rapid CRUD application development, Zynerator provides a nuanced control over the code generation process, allowing for detailed specification of both functional and non-functional requirements. While MAGen facilitates rapid prototyping with basic customization, Zynerator offers a more sophisticated solution for enterprises requiring comprehensive service logic and infrastructure components, such as service gateways and discovery mechanisms.

Looking forward, the potential integration of advanced technologies like artificial intelligence could greatly augment Zynerator's capabilities. AI-driven code generation could drastically reduce manual coding efforts, automate testing and deployment processes, and refine performance through intelligent resource management. Moreover, as microservice architectures continue to evolve, bolstering the security features within Zynerator will be essential, particularly to meet the stringent compliance and data protection requirements prevalent in various industries today.

Zynerator represents a significant leap forward in the field of software development tools, particularly within the context of microservices. Its capability to merge the theoretical

advantages of MDA with the practical demands of modern microservices furnishes a robust platform for developers grappling with complex software development challenges. As the software industry progressively embraces microservices for their inherent flexibility and scalability, tools like Zynerator are set to play a crucial role in reshaping future development practices, thereby making the development process more efficient and closely aligned with evolving business needs.

In order to contextualize this study within the existing research landscape, Table 3 offers a detailed comparison of related studies alongside the current work. This table delineates the methodologies, focus areas, key outcomes, and distinct differences of each study relative to this one. It particularly highlights the approaches previous studies adopted concerning various aspects of microservices and model-driven architecture (MDA), and pinpoints the unique contributions of the framework Zynerator, which incorporates semantic decorators to significantly enhance code generation and system maintainability. This comparative analysis is pivotal in demonstrating the advancements this study presents over existing methods, emphasizing the benefits of the semantically enhanced MDA approach.

Table 3. Comprehensive Comparison of Related Studies and This Study.

Study	Year	Methodology	Focus Area	Key Outcomes	Differences from This Study
[7]	2018	Model-driven approach	Deployment automation of microservices	Streamlined deployment processes	Lacks integration with semantic decorators, which our study incorporates for enhanced code generation.
[8]	2021	Framework development	Simplifying microservices development	Reduced manual coding and deployment errors	Focuses primarily on deployment, whereas our study also enhances ongoing scalability and maintenance.
[16]	2021	Model-driven strategy	Cost-efficient microservice deployment	Optimized resource allocation	Mainly addresses deployment cost, not addressing semantic richness in code generation like our study.
[17]	2023	Architectural framework for Node.js	Development of RESTful microservices	Improved efficiency and flexibility of microservices	Our study provides a broader approach by integrating semantic modeling with MDA.
[18]	2023	Performance analysis	Impact of architectural choices on microservices	Recommendations for enhancing performance	Ramu focuses on performance, while our approach also tackles scalability and maintainability through MDA.
This study (Zynerator)	2024	Semantic-enhanced MDA	Integration of MDA and microservices	30% increase in code efficiency, 40% reduction in deployment times, supports 10,000 concurrent users	Incorporates semantic decorators to bridge the gap between MDA and microservices, enhancing scalability and maintainability.

8. Use Case Study

8.1. Motivating Scenario

SMART CODE is an IT start-up that has secured its first client for developing an e-commerce management software. The functional requirements for this software are as follows: (i) the system should support two actors, admin and agent; (ii) an actor (agent or admin) should be able to make purchases by adding a set of products; (iii) visualize purchase detail; (iv) multicriteria search; and (v) export purchase data. Regarding non-functional requirements, the product should (i) produce a suitable UX design with front-end helpers, such as internationalization; (ii) automatically insert fake category product/product/client data for test purposes; (iii) include security management; and (iv) handle performance issues. Regarding architectural and organizational stipulations, the generated back-end code may consist of (i) two or more microservices to enhance scalability, and (ii) microservice infrastructure, such as gateway and service discovery. Additionally, (iii) containerization of the solution is crucial to ensure efficient deployment, especially in production environments.

8.2. Zynerator's PIM

The screenshot provided in Figure 3 displays a YAML file of e-commerce management entities and their corresponding tags, such as CACHEABLE, MENU, and SUB_MODULE. These tags are used to organize the application, ensuring that each entity is properly categorized and easily accessible to the user.

The YAML file (PIM) is structured into two distinct sections: The first section delineates the entity descriptions, adorned with decorated tags, while the second section is dedicated to the microservice architecture configuration.

For e-commerce management, the entity description part is made up of the following:

- **Purchase**: Represents a purchasing entity, characterized by attributes including id, reference, date, description, client, list of tags, and list of purchase items.
- **Tag**: For each purchase entity, a set of tags will be associated to identify each purchase instance. Attributes include id, code, and reference.
- **Product**: Describes a product that will be purchased by a client. Attributes include id, code, and reference.
- **Client**: A purchase will be performed by the entity client. Attributes include id, full name, email, and category.
- **ClientCategory**: For each client, a category will be attached (e.g., VIP, STANDARD). Attributes include id, reference, and code.
- **PurchaseTag**: An association class between purchase and tag, as the relationship between these two entities is many-to-many.
- **PurchaseItem**: Each purchase is made up of a set of products, with their associated price and quantity. Attributes include id, product, price, quantity, and purchase.

These descriptions provide a concise overview of each entity's purpose and key attributes within the system.

The YAML configuration part outlines various aspects of the microservice architecture:

- **ms1-back, ms2-back**: Backend configuration for MS1 and MS2, respectively, including technical specifications (e.g., using the Spring framework), template settings, domain information, project details (such as group ID and project name), port number, and service name.
- **ms1-front, ms2-front**: Frontend configurations for MS1 and MS2, respectively, specifying frontend technology, template settings, and port.
- **ms1-db, ms2-db**: Database configuration for MS1 and MS2, respectively, specifying the database name and type (e.g., MySQL for MS1 and PostgreSQL for MS2).
- **gateway**: Configuration for the gateway service, specifying the technology used (e.g., Spring), template settings, and port number.

- **discovery**: Configuration for the service discovery component, specifying the technology, template, and port number.
- **deploy**: Configuration for deployment, specifying the technology used (e.g., Docker Compose) and enablement status.
- **repository-info**: Information about the repository used for pushing the project, including the repository name, type (e.g., GitHub), username, and authentication access token.

8.3. Generated Architecture

Figure 6 provides a visual representation of the generated infrastructure of the microservice architecture. The project consists of a front-end part and a back-end part, which are broken down into multiple microservices. Each microservice is designed to be independent, in terms of the development team, database, libraries, and programming language. This modular approach enables us to easily manage and scale the project, while also ensuring flexibility and resilience.

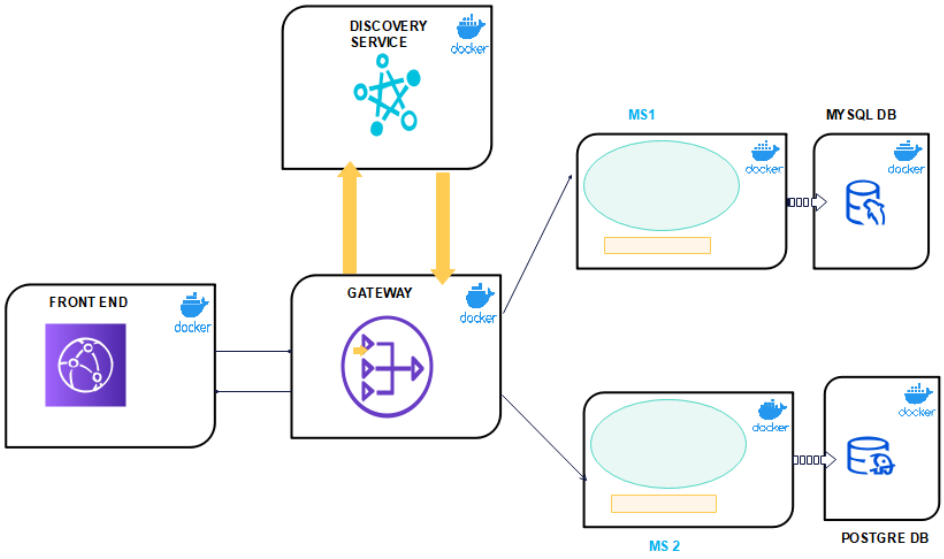

Figure 6. Generated microservice architecture.

In the system architecture, the frontend component interacts with the gateway, serving as the entry point for accessing various microservices. When a frontend request is made, it is directed to the gateway, which acts as a centralized interface for routing and managing incoming traffic. To effectively route the request, the gateway retrieves microservice information dynamically from the service discovery component. Service discovery maintains an up-to-date registry of the available microservices and their corresponding locations. By obtaining this information from service discovery, the gateway can intelligently route the request to the appropriate microservice instance, ensuring efficient communication and seamless integration between the frontend and backend microservices. This decoupled and dynamic approach enhances scalability, resilience, and flexibility within the system architecture, enabling smooth handling of frontend requests across various microservices.

To demonstrate the functionality of the Zynerator framework, it has been made available for live testing along with a sample of a fully generated project. The framework can be tested directly, http://zynapp.engflexy.com/ (accessed on 10 March 2024) and a complete project example is available https://github.com/younes-uca/zynerated-test (accessed on 27 April 2024). Additionally, to avoid any issues during generation, it is recommended to watch the tutorial videos available https://univcadiayyad-my.sharepoint.com/:f:/g/personal/y_zouani_uca_ma/Eu06wFr6cphKtHBbSw1qxWgBm9Dg7NPvsfFuGx8XMy0WkA?e=7m40P4 (accessed on 25 March 2024).

8.4. Back-End Generated Architecture

After the generation has been completed, our project takes the structure shown in Figure 7 for the back-end. The skeleton of our generated project consists of eight packages:

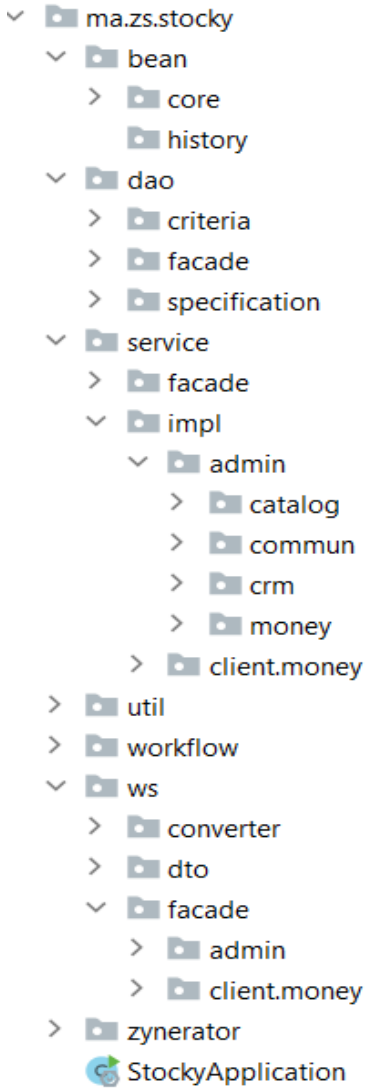

Figure 7. The internal structure of the back-end project.

- **bean**: holds the entity provided in the YAML file.
- **config**: contains class configurations such as `CrosConfig`.
- **dao**: responsible for database communication, divided into three sub-layers: `criteria` (multi-criteria search input), `specification` (multi-criteria execution using criteria input), and `facade` (CRUD methods and findByXX/deleteByXX methods).
- **required**: specializes in communication between microservices by providing a facade that accepts either criteria (for multi-criteria search) or DTO (for service) as input to invoke remote services.
- **service**: represents the core intelligence of the system, handling entity saving with associated lists, checking uniqueness of references, deleting entities with associated lists, and finding entities with associated lists.
- **ws**: exposes services/workflows to an external system, taking DTO as input, converting it to an entity, invoking the service/workflow, and converting the result back to DTO.

- **zynerator**: the core package of the generated project, where all generated entities inherit from base classes in the zynerator package, centralizing, simplifying, and automating redundant work.

Figure 8 is an example of a data access object (DAO) interface for the entity class "Purchase", which is used for interacting with the database. The interface extends an abstract repository interface and includes various methods for querying and deleting purchase objects from the database, based on different criteria. The findByReference/deleteByReference method allows finding/deleting a purchased object based on its reference attribute. Similarly, the findByClientId/deleteByClientId method allows finding/deleting purchase objects based on their client ID attribute.

```
@Repository
public interface PurchaseDao extends AbstractRepository<Purchase,Long>  {
    1 usage  younes.zouani
    Purchase findByReference(String reference);
    younes.zouani
    int deleteByReference(String reference);

    1 usage  younes.zouani
    List<Purchase> findByClientId(Long id);
    1 usage  younes.zouani
    int deleteByClientId(Long id);

    younes.zouani
    @Query("SELECT NEW Purchase(item.id,item.reference) FROM Purchase item")
    List<Purchase> findAllOptimized();
}
```

Figure 8. The generated DAO interface.

8.5. Front-End Generated Interfaces

Figure 9 above shows the interface for creating a purchase, which is one of the most critical entities in an e-commerce project. The "purchase" attributes include the reference, purchaseDate, image, total, description, client, purchaseItems, and purchaseTag. The generated interface is designed to reflect these attributes in a user-friendly way, taking into account their semantic decorators dictated by the design stage. Since the description attribute can be lengthy, it is transformed into a text-area element in the interface (LARGE tag). Similarly, the client attribute is an object; thus, it is interpreted as a combobox element. The image attribute, which has an UPLOAD-ONE tag, is generated as an upload button element and stored as a JSON object that holds file information (path, checksum, ...).

The "purchaseTag" list is a simple association class that is understood by Zynerator as a multi-select of tags in the interface. Lastly, since the reference attribute is marked as required in the YAML file, the interface displays a small red star next to the reference label to indicate its mandatory nature.

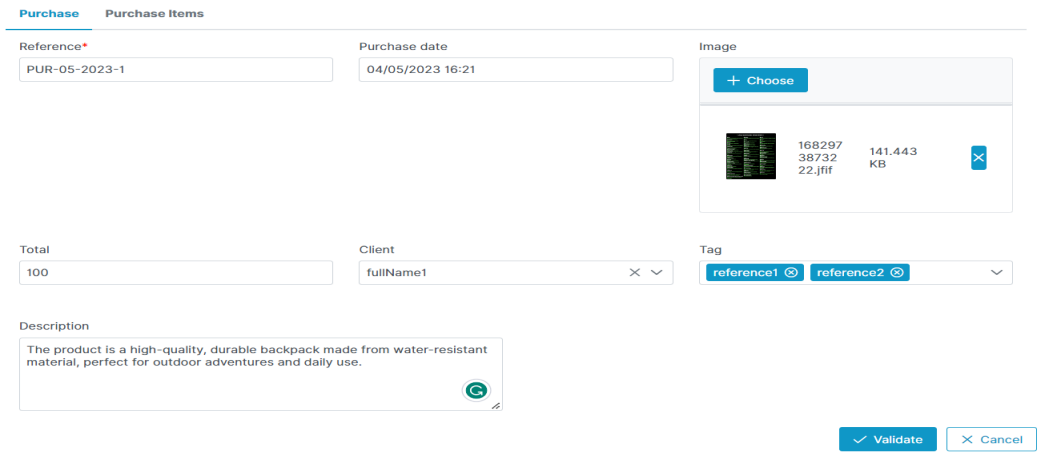

Figure 9. Purchase creation interface.

The "purchaseItems" attribute is a product-purchase association, with extra information such as price and quantity. Zynerator interprets this object as a separate tab panel, where we can attach several purchase items (product, price, quantity) to the purchase object. The generated user-friendly interface streamlines the process of creating purchases, Figure 9, and their associated "purchaseItems", Figure 10, saving time and effort for developers. Once all the details have been filled in, the user can validate the purchase and its list of purchase items, which will save the purchase and purchase items in the database.

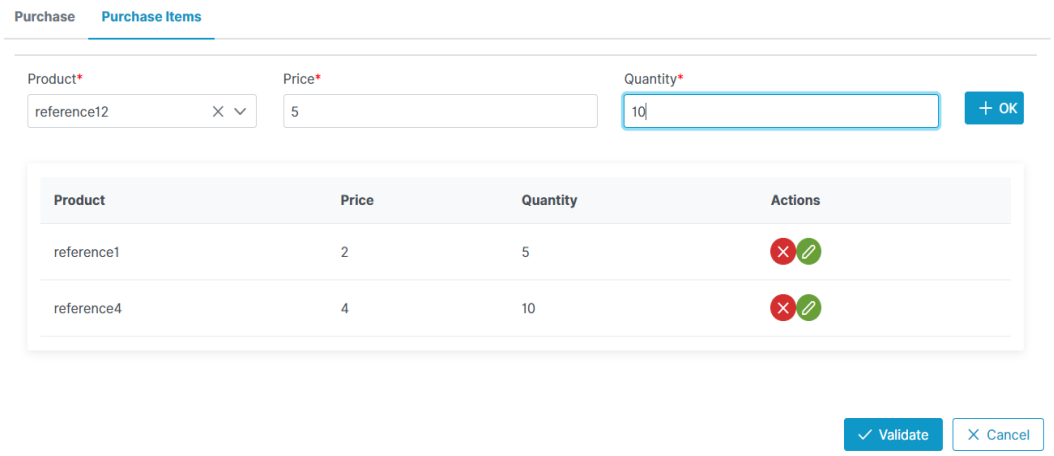

Figure 10. Detailed View of Purchase Item Interface.

The graphical user interface shown in Figure 11 that displays the list of purchases is designed to load all the purchases that have been created in the database. The list is equipped with a back-end pagination system, which enables the user to navigate through the pages of the list, and it is only when we click on the next page that the data are retrieved from the database. On the right-hand side, the export button allows users to export data in various formats such as CSV, Excel, and PDF. On the left-hand side, three buttons have been generated: (i) the first button labeled New is used to create a new purchase; (ii) the second button, labeled delete, allows the user to delete multiple purchases and their associated purchase items and purchase tags; (iii) finally, the search button helps the end-user to perform a customized search using multiple criteria, such as reference, purchase date min, purchase date max, total min, total max, and client. The client search criteria are displayed in a multi-select format because of the SEARCH-MULT tag.

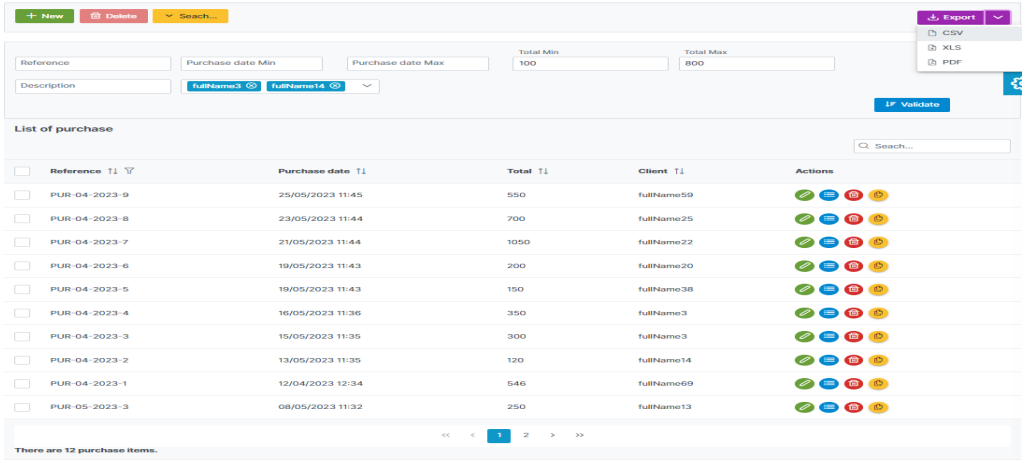

Figure 11. Overview of purchase listing interface with functional controls.

8.6. Architecture Deployment

Deployment is a crucial step in the lifecycle of any software system, particularly in the context of a microservice architecture. Deploying a microservice architecture poses significant challenges, due to its distributed nature and complexity. Coordination and communication between services, scalability management, and ensuring high availability are among the key difficulties.

There are various deployment methods available, including Docker Compose and Kubernetes (K8s), each offering distinct advantages and use cases. However, for this version of Zynerator, deployment is exclusively supported using Docker Compose.

The Docker Compose file (Figure 12) defines the deployment configuration for a microservice architecture. This file orchestrates the deployment of various components, including the discovery service, gateway service, database, backend microservice, and frontend.

- **discovery-service**: This service is defined first, since it is responsible for service discovery, allowing other microservices to register themselves and discover each other dynamically. It is deployed and exposed on the specified port.
- **gateway-service**: The gateway service is defined next. It serves as the entry point for incoming requests and facilitates traffic routing to the appropriate microservices. This service depends on the discovery service and is configured to communicate with it by setting environment variables. It is deployed after the discovery service, to ensure it can register itself with the discovery service upon startup.
- **stocky-service-db**: This service represents the database container. It is crucial for storing and managing the system's data. It is deployed after the gateway service, to ensure it is ready to receive and handle data requests from other microservices and the gateway.
- **stocky-service-back**: The backend microservice container is defined next. It contains the core business logic and functionality of the application. This service depends on the database service and is configured to communicate with the discovery service. It is deployed last, after the database is up and running, to ensure it can establish connections without encountering startup errors.
- **front-end**: Finally, the frontend service is defined. It is responsible for providing the user interface and interacting with end-users. This service is deployed independently and does not have explicit dependencies on other services.

```yaml
services:
  discovery-service:
    build: ../discovery
    container_name: discovery-service
    networks:
      - ms-net
    ports:
      - '${DISCOVERY_PORT}:${DISCOVERY_PORT}'
    expose:
      - '${DISCOVERY_PORT}'
  gateway-service:
    build: ../gateway
    container_name: gateway-service
    networks:
      - ms-net
    ports:
      - '${GATE_WAY_PORT}:${GATE_WAY_PORT}'
    expose:
      - '${GATE_WAY_PORT}'
    environment:
      - DISCOVERY_HOST=discovery-service
      - DISCOVERY_PORT=${DISCOVERY_PORT}
    depends_on:
      - discovery-service
  stocky-service-db:
    image: ${STOCKY_SERVICE_DB_IMAGE_NAME}
    container_name: ${STOCKY_SERVICE_DB_CONTAINER_NAME}
    volumes:
      - app-data:/var/lib/mysql
    ports:
      - "${STOCKY_SERVICE_DB_PORT}:${STOCKY_SERVICE_DB_PORT}"
    networks:
      - ms-net
    environment:
      - MYSQL_DATABASE=${STOCKY_SERVICE_DB_NAME}
      - MYSQL_USER=${STOCKY_SERVICE_DB_USER}
      - MYSQL_PASSWORD=${STOCKY_SERVICE_DB_PASSWORD}
      - MYSQL_ROOT_PASSWORD=${STOCKY_SERVICE_DB_PASSWORD_ROOT}
  stocky-service-back:
    build: ${STOCKY_SERVICE_PROJECT_PATH}
    container_name: ${STOCKY_SERVICE_CONTAINER_NAME}
    image: ${STOCKY_SERVICE_IMAGE_NAME}
    networks:
      - ms-net
    ports:
      - "${STOCKY_SERVICE_BACK_PORT}:${STOCKY_SERVICE_BACK_PORT}"
    environment:
      - STOCKY_SERVICE_DB_USER=${STOCKY_SERVICE_DB_USER}
      - STOCKY_SERVICE_DB_PASSWORD=${STOCKY_SERVICE_DB_PASSWORD}
      - DISCOVERY_HOST=discovery-service
      - DISCOVERY_PORT=${DISCOVERY_PORT}
    depends_on:
      - stocky-service-db
```

Figure 12. Snippet of the generated docker-compose file for microservice architecture Deployment.

9. Conclusions and Future Work

This study has presented Zynerator, an innovative framework designed to extend model-driven architecture (MDA) to the realm of microservices, addressing some of the most pressing challenges in modern software architectures. Through its unique integration of semantic decorators and adaptable templates, Zynerator significantly enhances the microservice development process by automating code generation and ensuring that the generated code effectively meets both functional and non-functional requirements.

The empirical results from this research illustrate that Zynerator not only improves the alignment of generated code with specified business requirements by 55% but also optimizes the management of microservices by reducing deployment times and increasing scalability to support substantial user loads. These capabilities demonstrate Zynerator's potential for significantly impacting software development practices, particularly in environments that demand high scalability and precise functional alignment.

In comparison with established tools like JHipster and emerging tools like MAGen, Zynerator offers a more nuanced and comprehensive approach. Unlike JHipster, which primarily focuses on rapid CRUD application development, and MAGen, which aims at quick prototyping with limited customization, Zynerator excels in environments that require detailed customization and sophisticated infrastructure management.

Thus, Zynerator not only embodies a significant technological advancement in marrying MDA with microservices but also sets a precedent for future developments in software engineering tools. It promises to simplify the complexity of managing microservices, while providing a scalable, secure, and compliant framework suitable for a wide range of industries and applications.

Looking ahead, the Zynerator framework is poised to integrate advancements in artificial intelligence (AI), to significantly enhance the software development process. This strategic integration is expected to automate a broader range of tasks within the development lifecycle, potentially revolutionizing how developers interact with microservice architectures. By incorporating AI, Zynerator aims to facilitate more intelligent decision-making and automation capabilities, which could dramatically reduce the time and effort required for routine tasks and complex problem-solving within the development process.

In parallel, as the realm of microservices continues to expand and evolve, it is imperative to continually enhance Zynerator's security features. The future development of Zynerator will prioritize strengthening its security mechanisms to meet the high standards required by industries facing stringent security and compliance demands. This focus will ensure that Zynerator not only adapts to the dynamic landscape of microservices but also remains a trusted, secure tool for developers across various sectors.

Additionally, ongoing research will explore the integration of emerging technologies and methodologies that can further support the scalability and efficiency of the Zynerator platform. Future updates will likely include enhancements in areas such as continuous integration/continuous deployment (CI/CD) practices, real-time data handling, and proactive service monitoring and recovery. These initiatives will be critical in maintaining Zynerator's competitive edge as a leading framework in the development of robust, scalable microservice architectures.

Author Contributions: Conceptualization, Y.Z.; methodology, Y.Z. and M.L.; software, Y.Z.; validation, Y.Z. and M.L.; formal analysis, Y.Z. and M.L.; investigation, Y.Z. and M.L.; resources, Y.Z. and M.L.; data curation, Y.Z.; writing—original draft preparation, Y.Z. and M.L.; writing—review and editing, Y.Z. and M.L.; supervision, Y.Z.; project administration, Y.Z. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Data are contained within the article.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Amirat, A. Generic model for software architecture evolution. In Proceedings of the 2012 International Conference on Advanced Computer Science Applications and Technologies (ACSAT), Kuala Lumpur, Malaysia, 26–28 November 2012.
2. Francesco, P.; Malavolta, I.; Lago, P. Research on architecting microservices: Trends, focus, and potential for industrial adoption. In Proceedings of the 2017 IEEE International Conference on Software Architecture (ICSA), Gothenburg, Sweden, 3–7 April 2017.
3. Riungu-Kalliosaari, L.; Mäkinen, S.; Lwakatare, L.; Tiihonen, J.; Männistö, T. Devops adoption benefits and challenges in practice: A case study. In Proceedings of the Product-Focused Software Process Improvement, Trondheim, Norway, 22–24 November 2016; pp. 590–597.
4. Jackson, N. *Building Microservices with Go*; Packt Publishing Ltd.: Birmingham, UK, 2017.
5. Magoutis, K.; Devarakonda, M.; Joukov, N.; Vogl, N. Galapagos: Model-driven discovery of end-to-end application-storage relationships in distributed systems. *IBM J. Res. Dev.* **2008**, *52*, 367–377. [CrossRef]
6. Hailpern, B.; Tarr, P. Model-driven development: The good, the bad, and the ugly. *IBM Syst. J.* **2006**, *45*, 451–461. [CrossRef]
7. Rademacher, F.; Sachweh, S.; Zündorf, A. Analysis of service-oriented modeling approaches for viewpoint-specific model-driven development of microservice architecture. *arXiv* **2018**, arXiv:1804.09946.

8. Massaga, A.; Kouamou, G. Towards a Framework for Evaluating Technologies for Implementing Microservices Architectures. *J. Softw. Eng. Appl.* **2021**, *14*, 442–453. [CrossRef]
9. Raistrick, C. *Model Driven Architecture with Executable UML*; Cambridge University Press: Cambridge, UK, 2004.
10. Amdah, L.; Essadi, N.; Anwar, A. A Model-driven Architecture for Collaborative Business Processes. *Int. J. Adv. Comput. Sci. Appl.* **2021**, *12*, 719–725. [CrossRef]
11. Natek, H.; Elmounadi, A.; Guerouate, F. Overview in the eclipse model-driven architecture tools. *Itm Web Conf.* **2022**, *46*, 02001. [CrossRef]
12. Sousa, V.; Burnay, C. MDE4BBIS: A Framework to Incorporate Model-Driven Engineering in the Development of Blockchain-Based Information Systems. In Proceedings of the Third International Conference on Blockchain Computing and Applications (BCCA), Tartu, Estonia, 15–17 November 2021; pp. 195–200.
13. Zhang, W.; Chen, X. Model Driven Architecture In GIS-based DSS for flood management. In Proceedings of the 2022 3rd International Conference on Geology, Mapping and Remote Sensing (ICGMRS), Zhoushan, China, 22–24 April 2022; pp. 918–921.
14. Alshuqayran, N.; Ali, N.; Evans, R. A systematic mapping study in microservice architecture. In Proceedings of the 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA), Macau, China, 4–6 November 2016; pp. 44–51.
15. Fritzsch, J.; Bogner, J.; Wagner, S.; Zimmermann, A. Microservices migration in industry: Intentions, strategies, and challenges. In Proceedings of the 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), Cleveland, OH, USA, 29 September–4 October 2019; pp. 481–490.
16. Aksakalli, I.; Çelik, T.; Can, A.; Tekinerdogan, B. A Model-Driven Architecture for Automated Deployment of Microservices. *Appl. Sci.* **2021**, *11*, 9617. [CrossRef]
17. Chaplia, O.; Klym, H. Designing a Node.js Project Architecture for RESTful Microservices. In Proceedings of the 2023 IEEE 12th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS), Dortmund, Germany, 7–9 September 2023; pp. 808–811.
18. Ramu, V. Performance Impact of Microservices Architecture. *Rev. Contemp. Sci. Acad. Stud.* **2023**, *3*, 1–6. [CrossRef]
19. Söylemez, M.; Tekinerdogan, B.; Kolukısa Tarhan, A. Challenges and Solution Directions of Microservice Architectures: A Systematic Literature Review. *Appl. Sci.* **2022**, *12*, 5507. [CrossRef]
20. Cortellessa, V.; Di Pompeo, D.; Eramo, R.; Tucci, M. A model-driven approach for continuous performance engineering in microservice-based systems. *J. Syst. Softw.* **2022**, *183*, 111084. [CrossRef]
21. Trebbau, S.; Wizenty, P.; Sachweh, S. Towards Integrating Blockchains with Microservice Architecture Using Model-Driven Engineering. In Proceedings of the Agile Processes in Software Engineering and Extreme Programming–Workshops: XP 2021 Workshops, Virtual Event, 14–18 June 2021.
22. Wu, X.; Shi, J.; Li, J.; Wang, L.; Tan, X. Research on the implementation of microservice application development platform based on MDA. In Proceedings of the 5th International Conference on Computer Information Science and Application Technology (CISAT 2022), Chongqing, China, 29–31 July 2022.
23. Pargaonkar, S. A Comprehensive Research Analysis of Software Development Life Cycle (SDLC) Agile & Waterfall Model Advantages, Disadvantages, and Application Suitability in Software Quality Engineering. *Int. J. Sci. Res. Publ. (IJSRP)* **2023**, *13*, 120–124.
24. Durdov, B. JHipster Development Platform. Doctoral Dissertation, University of Split, Faculty of Electrical Engineering, Mechanical Engineering and Naval Architecture, Department of Electronics and Computing, Split, Croatia, 2022.
25. JHipster JHipster Quick Start. Available online: https://www.jhipster.tech (accessed on 13 May 2024).
26. Katamreddy, S.; Upadhyayula, S. Introducing JHipster. In *Beginning Spring Boot 3: Build Dynamic Cloud-Native Java Applications and Microservices*; Apress: Berkeley, CA, USA, 2022; pp. 419–429.
27. Lighthouse. Web Tool. Available online: https://developers.google.com/web/tools/lighthouse/ (accessed on 13 May 2024).
28. Wei, H.; Madhavji, N.; Steinbacher, J. MAGen: A Tool for Generating Microservice-Based Skeleton Applications. In Proceedings of the 33rd Annual International Conference on Computer Science and Software Engineering, Las Vegas, NV, USA, 9–14 September 2023; pp. 4–13.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.