IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 6 June 2025, accepted 10 July 2025, date of publication 15 July 2025, date of current version 22 July 2025.

Digital Object Identifier 10.1109/ACCESS.2025.3589285

RESEARCH ARTICLE

Microservices-Driven Automation in Full-Stack Development: Bridging Efficiency and Innovation With FSMicroGenerator

SAMIRA KHALFAOUI[1], HAFIDA KHALFAOUI[2], AND ABDELLAH AZMANI[1]

[1]Intelligent Automation and BioMedGenomics Laboratory, Faculty of Science and Technology of Tangier, Abdelmalek Essaâdi University, Tétouan 93000, Morocco

[2]LIMATI Laboratory, Polydisciplinary Faculty, Sultan Moulay Slimane University, Beni Mellal 23000, Morocco

Corresponding author: Samira Khalfaoui (samira.khalfaoui@etu.uae.ac.ma)

This work was supported in part by the Ministry of Higher Education, Scientific Research, and Innovation, and the Digital Development Agency (DDA); and in part by the National Center for Scientific and Technical Research (CNRST) of Morocco under the Smart Digital Logistic Services Provider (DLSP) Project AL Khawarizmi Artificial Intelligence (AI)-Program.

ABSTRACT The development of modern web applications presents major challenges due to increasingly complex architectures, scalability requirements, and the need to reduce time to market. These challenges are exacerbated by the shortage of talent, making the implementation of modern architectures more difficult for many companies. In response, this article presents FSMicroGenerator, an innovative code generation tool based on a low-code approach, which exploits UML class diagrams and templates to generate multilingual, full-stack web applications based on a microservices architecture. FSMicroGenerator automates the entire process of creating IT solutions, from development to deployment, reducing technical barriers and technical debt, and enabling teams to focus on business innovation while ensuring compliance with industry standards and best practices. By facilitating the adoption of DevSecOps practices, it improves collaboration between the development and operations teams, ensuring that security is integrated at every stage of the development process. This enables continuous management of the application lifecycle, faster and more reliable releases, and proactive identification and mitigation of security vulnerabilities. FSMicroGenerator also stands out for its ability to create a catalog of ready-to-use functional blocks, enabling their reuse and integration into other solutions. In addition, it offers a secure ecosystem, protecting companies' assets against the risks of data exposure and leakage thanks to a robust architecture and integrated security mechanisms. As part of our open innovation approach, we plan to make FSMicroGenerator open source, to broaden its adoption and enable the community to contribute to its continuous improvement.

INDEX TERMS Automatic code generation, DevSecOps, full stack, low-code approach, microservices, multilingual, software development, templates, UML class diagram.

I. INTRODUCTION

With rapid advances in technology and the growing demand for sophisticated IT systems, expectations for software functionality continue to rise [1]. Users and companies demand advanced solutions that meet complex needs and provide a smooth, efficient user experience. This demand increases the quality, quantity, and size of the code produced. However, the massive amount of code that needs to be developed and maintained daily is becoming extremely overwhelming [2]. Developers find themselves devoting a disproportionate amount of their time to repetitive and redundant tasks at the expense of innovation and solving complex problems. In 2002, a study by the National Institute of Standards and Technology revealed that developers spend almost 80% of their time debugging and fixing bugs, with approximately 40 to 50% of this effort devoted to rework that could have been avoided [3]. Writing code manually for each

The associate editor coordinating the review of this manuscript and approving it for publication was Fabrizio Messina.

feature becomes not only tedious, but also prone to errors that can compromise quality, delay delivery of the final product, and increase costs.

A. RESEARCH MOTIVATION

The software engineering community has long recognized the need for standardized modeling paradigms to optimize software development and maintenance. Early paradigms such as Model-Driven Architecture (MDA), Model-Driven Development (MDD), Model-Driven Engineering (MDE), Model-Based Architecture (MBA), and Model-Driven Software Engineering (MDSE) have positioned modeling as a fundamental approach to improving development efficiency [4].

Despite their adoption, these modeling paradigms continue to face challenges related to complexity, scalability, and the efficiency of model transformations. Moreover, they depend heavily on modeling expertise, making them less accessible to non-technical users. This requirement contradicts the growing talent shortage in software engineering, as businesses increasingly seek solutions that enable faster development with minimal technical expertise. To mitigate these limitations, development environments have incorporated Template-Based Code Generation (TBCG) mechanisms, such as Microsoft's Text Template Transformation Toolkit (T4) and Apache Velocity [5].

Since the 1990s, TBCG has simplified code generation by promoting reuse and automation. It relies on predefined templates containing static code and dynamic meta-code, processed by a template engine to generate source code. This technique, based on the principle of "code once, implement everywhere", has gained popularity due to its effectiveness and ease of use [6].

TBCG improves MDE by offering a more structured and reusable approach to code generation. Although MDE relies on model-to-text transformations to generate code from platform-independent models refined into platform-specific models [7], TBCG improves automation by using predefined templates. This ensures higher abstraction and more effective bridging between modeling and programming [7].

Despite its advantages, TBCG has several limitations. It is often restricted to specific domains such as User Interfaces (UI), Create, Read, Update, Delete (CRUD) operations, and Application Programming Interface (API) generation. It also depends on specific languages or frameworks and relies on complex, manually designed templates. These factors limit its accessibility for non-technical users [6].

To address these challenges, Low-Code/No-Code (LCNC) platforms have gained popularity as a solution to the increasing demand for applications amid a shortage of skilled developers. LCNC platforms enable the rapid creation of efficient and scalable applications with minimal coding effort, introducing the concept of citizen developers in application development [8]. These platforms abstract complexity by allowing users to design applications through graphical interfaces and predefined components, significantly reducing the need for deep programming knowledge.

Furthermore, the integration of Generative AI (GenAI) in software development enhances flexibility and automation. Key capabilities include code generation, which produces complete code blocks from text descriptions; code completion, suggesting fragments to speed up writing; and code translation, converting code from one programming language to another. GenAI also supports code refinement to improve readability and efficiency, code summarization that provides concise explanations of complex modules, and defect detection to identify errors and redundant code through clone detection [9], [10], [11].

Despite their advantages, LCNC platforms and GenAI face significant limitations. LCNC solutions can lead to vendor lock-in [12], limited customization, and scalability issues in high-load environments [13]. Additionally, "shadow IT" emerges when employees, facing obstacles in meeting their needs, adopt unauthorized tools and solutions that bypass official processes and systems. This can lead to technical debt, especially when non-developers build applications without proper governance [13], [14].

Similarly, GenAI tools depend on training data quality, making them prone to errors, incomplete solutions, and security risks [15]. Ensuring compliance with coding standards remains challenging [9], and Artificial Intelligence (AI) alone cannot replace human oversight in complex decision-making. Over-reliance on AI can also weaken the problem-solving skills of developers over time [16].

Addressing these challenges requires developing solutions that ensure flexibility, security, and scalability in software development. Existing approaches often struggle to balance customization, maintainability, and efficiency, making it essential to explore more adaptable and robust methodologies that enhance scalability and security.

These observations suggest that combining low-code paradigms with structured, template-driven approaches can provide a scalable and reliable alternative to address the limitations of GenAI and LCNC tools in web development. Such an approach mitigates these GenAI limitations by relying on deterministic, user-controlled templates, ensuring predictable outputs without dependence on training data quality.

B. MAIN CONTRIBUTIONS

In response to the limitations of existing approaches, researchers and developers have increasingly turned to code generation tools to meet business needs. These tools aim to automate development, improving efficiency and accessibility while reducing manual effort in creating complex applications. However, many existing solutions are language-specific or designed for monolithic architectures, leading to rigidity in maintenance and challenges in evolving toward modular solutions. Additionally, most tools focus solely on either the frontend or backend, making

full-stack development challenging. They often require time-consuming manual input, such as database tables, CRUD operation parameters, and HTML configurations, while lacking support for template import or modification, which limits their flexibility and adaptability.

To overcome these gaps, this paper presents FSMicroGenerator, a low-code tool that leverages UML class diagrams and templates to generate scalable, secure, and maintainable full-stack web applications based on a microservices architecture. The main contributions are:

- **A unified low-code framework:** We propose FSMicroGenerator's architecture for generating full-stack microservices applications from UML diagrams and templates, reducing development complexity.
- **Scalable, maintainable, and secure design:** We demonstrate how the generated architecture supports scalability and maintainability, while integrated DevSecOps practices enhance security.
- **Performance evaluation:** We present a comparative case study, supported by an industrial use case, showing improvements in development speed and architectural quality over manual methods.
- **Reusable component catalog:** We enable the generation of a catalog of modular, reusable microservices that can be integrated into future projects to reduce duplication and speed up development.

C. ORGANIZATION OF THE PAPER

The remainder of this paper is organized as follows. Section II reviews related work on code generation and their limitations. Section III details FSMicroGenerator architecture, while Section IV focuses on the structure and features of the applications it generates. Section V presents the evaluation methodology used to assess FSMicroGenerator. Section VI compares the results of manual and generator-based development using quantitative and qualitative metrics. Section VII analyzes the observed benefits and limitations of the tool and compares it to alternative approaches. Finally, Section VIII concludes the paper and outlines directions for future work.

II. LITERATURE REVIEW

Code generation techniques have attracted growing interest in recent years due to their potential to automate and accelerate software development. We have selected a specific set of works [1], [2], [6], [17], [18], [19], [20] for a detailed analysis, as these sources are seminal in the development of code generation tools, covering a broad spectrum of methodologies and application areas. This review highlights the main approaches, tools and techniques developed in this field, while highlighting their advantages and limitations.

Uyanik and Sahin [1] introduced an automated code generation tool to accelerate web application development, particularly for TBYS, which is an Enterprise Resource Planning system developed by the Turkish Management Sciences Institute (TÜSSİDE). The tool generates CRUD operations and UI components using T4 templates for the backend and Handlebars for creating HTML and JavaScript plugins for the frontend. These plugins define both the visual appearance and the functionality of interface elements. Additionally, JavaScript Object Notation (JSON) files are generated to facilitate communication between the frontend and backend services. The integration of these components follows the ASP.NET MVC architecture, and communication between the frontend and backend is handled via RESTful APIs.

In a separate work, Irfan Ullah and Irum Inayat [6] propose an approach for automatic code generation from UML class diagrams. The tool is implemented in Python using the blackboard architecture, where multiple knowledge sources process domain-specific information to generate code. After the class diagram is parsed, the extracted data is made available to the knowledge sources, each of which applies its respective templates to generate the corresponding code. These sources then generate the necessary code for backend components, such as CRUD business logic and basic UIs. Unlike modern full-stack solutions that provide distinct and modular frontend and backend architectures, this tool directly integrates the UI within the Python application, making it unsuitable for full-stack development in the contemporary sense.

Marco Livraghi [17] focuses on full-stack development within a monolithic architecture that integrates both the frontend and backend into a single application. The system uses Spring for the backend and AngularJS for the frontend, generating RESTful services, UI, and documentation with Swagger. It emphasizes producing readable, modifiable, and easy-to-use code. The tool takes inputs such as Java classes, XML models, or database schema to generate the necessary code and offers advanced security management with multi-role access control and form validation. However, its monolithic architecture can be restrictive for environments that would benefit from distributed architectures such as microservices.

Taking a different approach, Paolone et al. [18] present xGenerator, a tool for generating Java web applications based on the Model-View-Controller (MVC) pattern. The tool uses UML use case and class diagrams as input and applies MDA model transformations across the computation independent model, platform independent model, and platform specific model layers. It supports full development, from business logic to UI, using the object-oriented paradigm. The generated applications rely on frameworks like Hibernate for database management and Vaadin for graphical interfaces. However, this is not a full-stack solution, as the generated UI components are directly integrated into the Java application without clear separation. They remain basic and lack customization. Furthermore, the approach is monolithic and does not support microservices.

In another similar approach, but specific to the Laravel framework, a widely used PHP framework for building web applications, Mantas Ražinskas and Lina Čeponienė

[19] have proposed an MDA-based methodology for the automatic generation of Laravel code from UML diagrams. This research aims to simplify information system development with PHP Laravel by automating UML-to-Laravel code transformation, reducing development time and errors. The transformation process occurs in three stages: first, a platform-independent model is created using UML tools and exported in XMI format. Then, a model-to-model transformation generates a platform-specific model by applying transformation rules, including Laravel-specific stereotypes such as 'LaravelModel', 'LaravelController', and 'LaravelBlade', which define components like models, controllers, and views. Finally, a model-to-code transformation generates the corresponding PHP files using tools like Eclipse Acceleo, an open-source code generation tool that transforms models into code based on the MDA. This approach does not aim to generate the entire code, but rather to automate the transition from UML models to code as much as possible, while leaving developers free to complete the system manually.

From a different perspective, the study by Prasanthan et al. [2] presents Backend as a Service (BaaS), a tool that accelerates backend development by generating a pre-configured code base with CRUD functionality and authentication features. Users provide a JSON-based data model, specifying field types and constraints, and the tool outputs a Docker container, facilitating microservices integration and optimizing resource utilization. The tool was evaluated through an experiment that compared it with manual development, analyzing creation time, errors, code structure, and uniformity. Results showed that BaaS improved efficiency, reduced deployment errors via Docker, and generated uniform, maintainable code. However, BaaS is limited to backend generation and does not handle additional constraints on data fields (e.g., minimum length, maximum length). Although it encourages a microservice architecture, it lacks features for assembling services into a single application or managing communication between generated APIs.

While the majority of previous work focuses on specific application layers, Tesoriero et al. [20] offer a comprehensive solution for full-stack development of multi-layered web applications based on MDA. The tool automates the development of persistence, web services, and presentation layers from UML class diagrams, following a monolithic client-server architecture. It utilizes PHP (Propel ORM) for backend, REST APIs for communication, and Polymer for UI generation. Integration with Eclipse via Papyrus and TagML-based XML processing ensures interoperability with third-party systems. However, the tool is restricted to specific technologies (e.g., MySQL for persistence) and lacks support for non-relational databases or advanced frontend frameworks. Additionally, its monolithic structure limits scalability, while customization options remain insufficient for complex applications.

The analysis of the literature highlights that many code generation tools are tightly coupled to specific languages, limiting their adaptability across different technological environments. Additionally, they often require complex configurations, which can reduce the efficiency of automation. Most tools favor monolithic architectures, despite the advantages of microservices, and those that adopt microservices often lack mechanisms for service integration and communication, making them unsuitable for distributed systems. Moreover, while some approaches address both business logic generation and user interface creation, they cannot be considered full-stack solutions in the modern sense due to the lack of a clear separation between frontend and backend components or the absence of modular and scalable architectures. Furthermore, reliance on heterogeneous languages and frameworks increases technical debt, complicates maintenance, and slows down project evolution. These limitations underscore the need for more flexible tools, capable of supporting modern architectures and managing the full development cycle, from configuration to service integration and deployment.

III. FSMICROGENERATOR ARCHITECTURE

A. FUNCTIONAL ARCHITECTURE

1) MAIN FEATURES

The tool aims to simplify and accelerate full-stack application development through continuous delivery and deployment. Addressing the need for modularity, reusability, and scalability, it structures applications into three logical units inspired by microservices principles:

- **Package**: A base unit contains specific, autonomous functionalities reusable across contexts. It groups tightly coupled business units that are highly interdependent for a specific function and have no standalone meaning outside the package. Packages represent reusable functional blocks, not necessarily linked to a complete business workflow.
- **Module**: A combination of packages or other modules collaborating to cover a larger or complete business functionality. Entities within a module are less tightly coupled than those in a package but work together to achieve a business process.
- **Project**: A complete, ready-to-use application built by combining modules and packages for a specific need. Coupling between project entities is weak or non-existent, with independent entities collaborating via clearly defined interfaces.

This structure leverages Domain-Driven Design (DDD) to decompose systems into bounded contexts (packages and modules), aligning them with business domains. This facilitates creating microservices with clear boundaries, ensuring cohesion, scalability, and business alignment while balancing granularity and communication [21].

The adopted organization fosters a flexible, reusable, and evolving architecture. It enables building a catalog of

ready-to-use packages and modules, reducing development effort and errors. Extensibility is ensured, allowing new components to be added without impacting existing ones due to well-defined interfaces and minimal coupling, making solutions adaptable and scalable.

2) FUNCTIONAL ORGANIZATION

FSMicroGenerator is organized around three key functional components: template management, user interface, and code generation engine, all designed for intuitive use and scalability.

- **Template management:** Template management is the heart of our tool. It is responsible for defining, organizing and customizing predefined code models, called templates, which serve as the basis for generating packages, modules or complete projects. Each template is designed to be modular, meaning that it can be used independently or in combination with others to create complex solutions. FSMicroGenerator is based on two types of template:
  - *Dynamics templates:* These generate code snapshots for individual solution components. They include microservices templates, which produce all files needed to build and run an independent microservice, and frontend templates, which supply files to create the UI, specifically tailored for the Angular framework to interact with these microservices.
  - *Static templates:* These define the overall architecture and structure of the generated solution, acting as a predefined skeleton. They include basic microservices essential for application function, ensure standardization across solutions, accelerate development by preconfiguring architecture, and serve as a foundation for customization using dynamic templates.

  Dynamic and static templates are complementary. Static templates define the application's structural backbone (e.g., folder hierarchy, base services), while dynamic templates are customized based on the business model to generate contextual microservice code. This separation enables granular customization, promotes reuse, and ensures a clear division of concerns between architecture and functional logic.
- **User interface:** This is the main point of interaction between the developer and FSMicroGenerator. It's designed to be intuitive and accessible, even for beginners. It provides dedicated areas for managing packages, modules, and projects, as well as user, team, and role-based access management. The interface makes configuration, generation, and full lifecycle tracking easy, guiding users through each development stage with transparency and control. Additionally, multi-language support removes language barriers and improves the experience for international or multicultural teams.
- **Code generation engine:** The code generation engine converts user configurations into ready-to-use source code using predefined templates. It speeds up development while ensuring high-quality, standards-compliant code that meets business needs. The engine validates data consistency, resolves dependencies, and ensures schema compatibility. Generated code follows the MVC pattern for better readability and maintenance. The engine is extensible, allowing easy addition of new templates or features without major changes, supporting ongoing adaptation to new technologies and requirements.

3) PACKAGE LIFECYCLE

The package is the core unit of FSMicroGenerator. This section details its lifecycle, covering all stages from creation to pre-integration, as illustrated in Figure 1. Understanding this lifecycle is key to grasping how the tool manages generation and adaptation to meet user-defined business specifications with high-quality output.

- **Creation stage:** This phase marks the beginning of the package lifecycle. The user starts by providing basic configuration details such as the package name, database name, service port, and the responsible data entry team. Next, the business data model is defined using UML class diagrams, where the team specifies entities, their attributes, and relationships. UML is a widely adopted and well-known modeling language that ensures ease of learning and reduces design time [20], making it an ideal choice for user input. FSMicroGenerator supports various attribute types and constraints (e.g., required, unique, value limits, patterns, enumerations), along with standard associations like one-to-many or recursive links, enabling robust and scalable data structures. Based on this model, the system can automatically generate HTML components, including tables and forms, which are fully customizable. It adapts input types to appropriate UI elements (e.g., a date field becomes a date picker) and allows real-time preview and adjustment. Figure 2 illustrates a demonstrative example of data model definition, while Figure 3 presents the configuration interface for the generated table and form. Throughout the process, default validation rules enforce naming conventions and check for uniqueness and type compatibility. Once all elements are configured, the package is submitted for validation.
- **Validation stage:** Since the first phase involves semi-automatic steps and manual configuration, errors are likely. Typical issues include missing attribute declarations, incorrect types or names, and undeclared relationships between entities. To catch these, a validation user—familiar with UML and the business context—reviews the package for consistency with functional and technical requirements. If errors are found, the package is marked as ''In the process of being corrected'' and returned with clear instructions. Once corrected by the team, it is resubmitted and

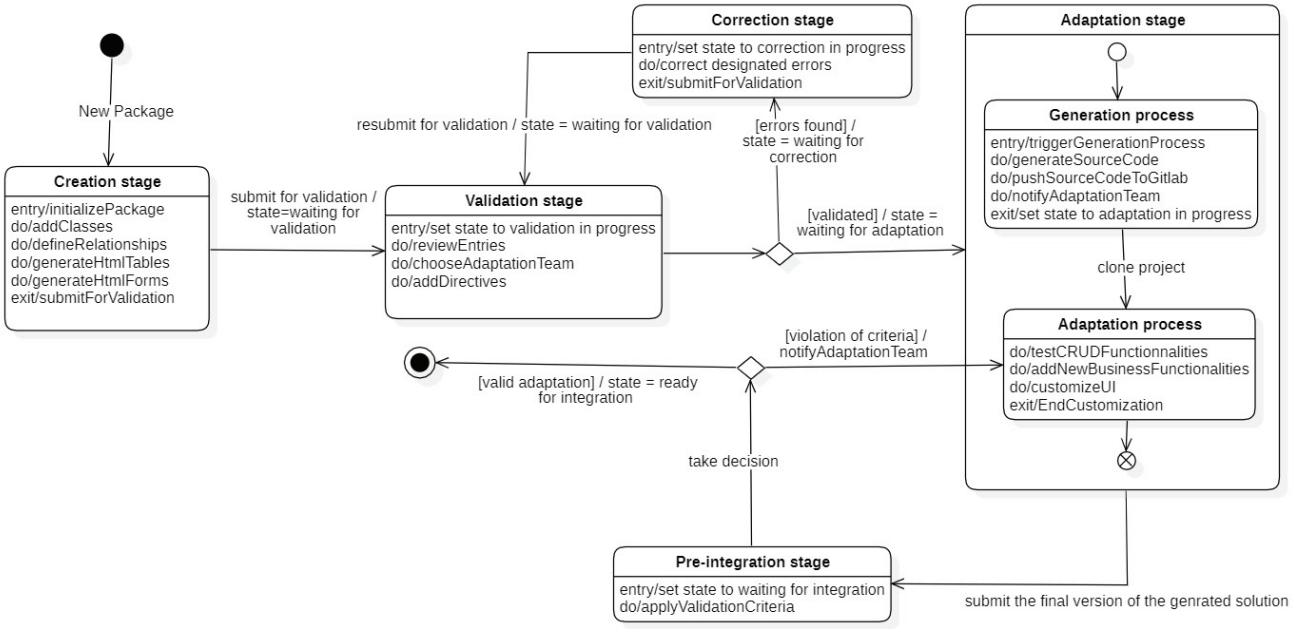


FIGURE 1. UML state diagram depicting the lifecycle of a package.

revalidated. If no errors are found, the package moves to the adaptation phase.

- **Adaptation stage:** The adaptation stage is where the conceptual model is transformed into executable code. This involves triggering the generation process, assigning the adaptation team, and performing tests and adjustments. Full details are provided in Section III-A4.
- **Pre-integration stage:** This stage marks the final phase in the package lifecycle, indicating that a package is ready to be incorporated into larger modules or projects. It serves to detect and resolve any remaining inconsistencies, ensuring that all adjustments comply with established standards and initial requirements. At this point, key validation criteria—such as code quality and structure, performance, resilience, and overall compliance—are considered. This validation step is optional: if the adaptation team has already verified these criteria upstream, the package can be directly marked as 'Ready for integration' to streamline the process. Once validated, the package is confirmed as complete, interoperable, and compatible with other components, requiring no major changes before integration.

4) GENERATION PROCESS

To ensure high-quality output aligned with user specifications, FSMicroGenerator employs a structured generation process, particularly during the *adaptation stage* of a package's lifecycle. This process transforms the user-defined conceptual model into a functional application through the following key steps:

a: STEP 1: CONFIGURATION EXTRACTION

The process begins when the adaptation team triggers the generation after validating the package definition. The code generation engine extracts the complete package configuration, including the data model and UI component specifications (tables, forms), defined during the *creation stage*.

b: STEP 2: SKELETON SETUP (STATIC TEMPLATES)

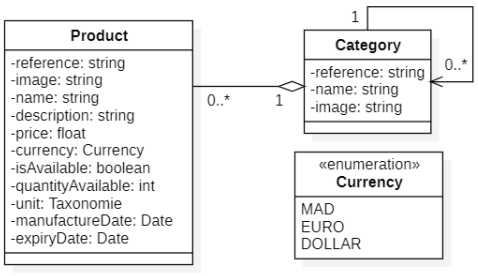
The engine loads relevant *static templates* to establish the foundational architecture and directory structure. This creates a predefined skeleton incorporating:

- API Gateway configuration.
- Base frontend application structure with common utilities (e.g., reusable components, pipes, directives, custom data validators).
- Base microservices structure with common utilities (e.g., caching, middlewares, monitoring, logging).
- Integration points for essential complementary microservices (e.g., member management, technical configuration, taxonomy).

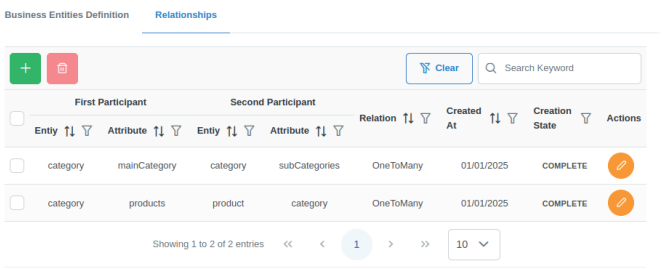
This step ensures structural standardization and integrates core functionalities.

c: STEP 3: BACKEND MICROSERVICES GENERATION (DYNAMIC TEMPLATES)

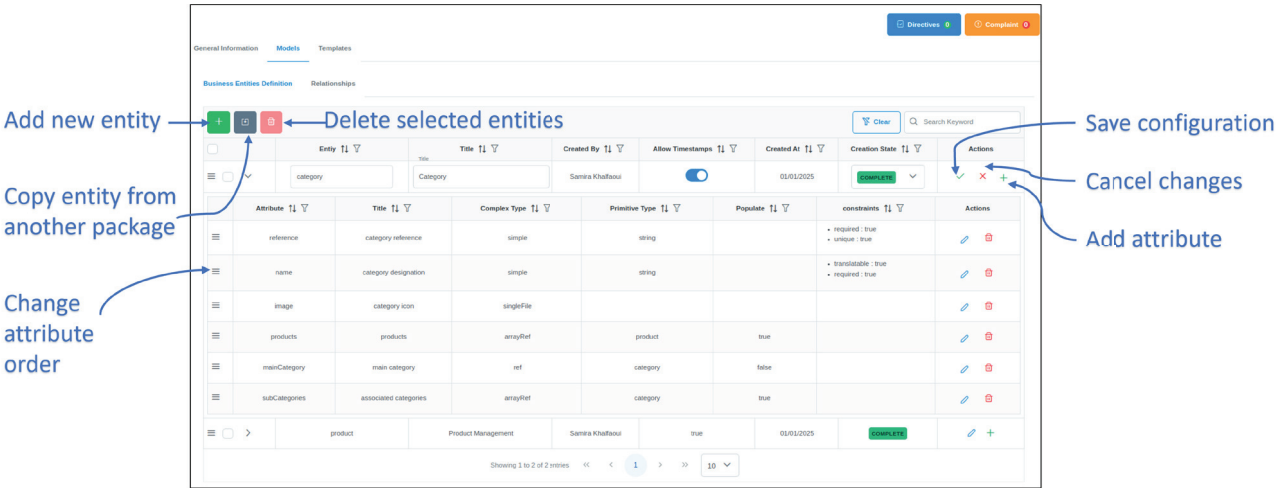
The engine iterates through each entity defined in the configuration. Using specialized *dynamic microservices templates*, it generates a set of backend code files for the main package microservice. Each generated file follows the naming pattern `${entityName}.<component>.js`,



(a) UML class diagram representing the business entities and their relationships, serving as the design-time input.



(b) UI listing the defined relationships between entities, specifying association types and participant attributes.



(c) UI for defining business entities, including attributes, data types, and constraints.

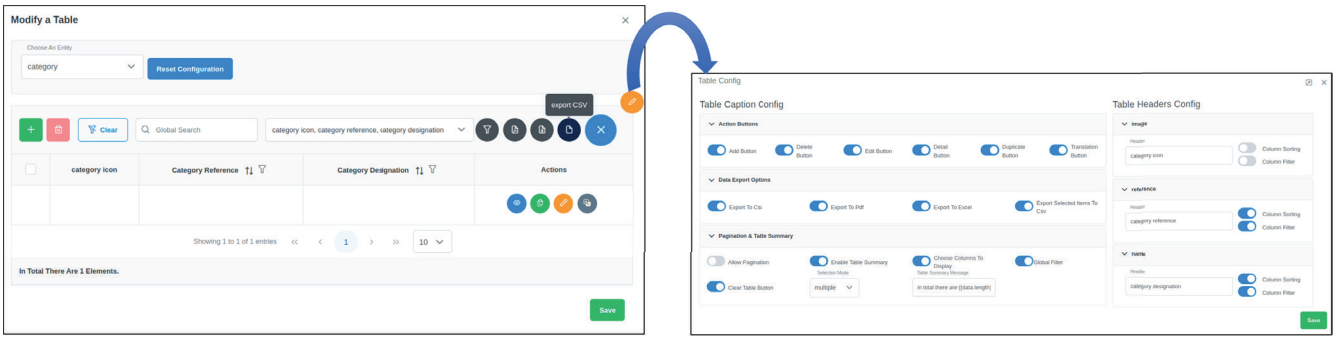
FIGURE 2. Overview of the data model definition process.

where <component> refers to the role of the file (e.g., model, repository, controller). The generation covers the following components:

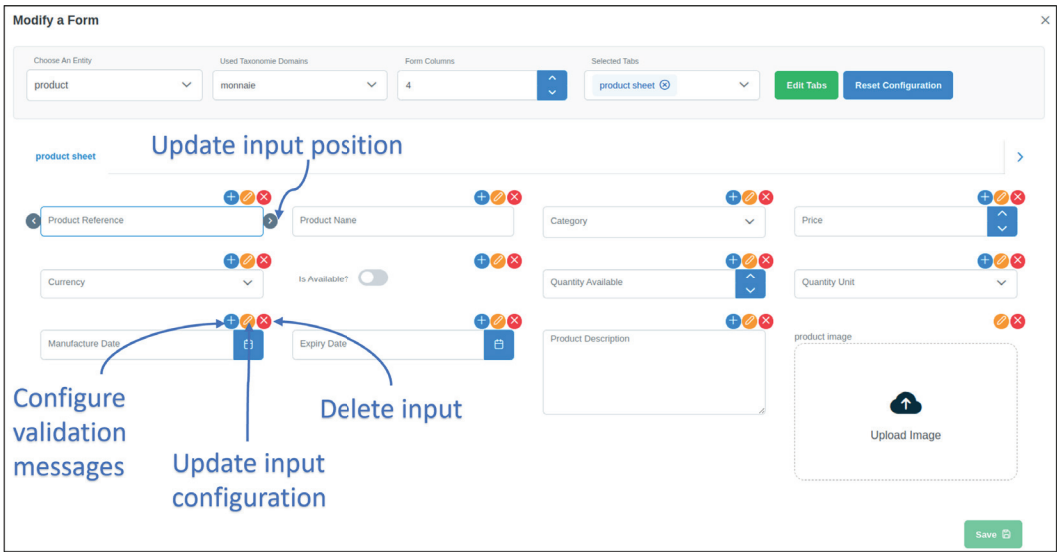
- **Model:** Generates the data model based on entity definitions. Listing 1 presents a simplified template snapshot used for generating the model file. This template also incorporates logic to handle defined relationships between entities automatically.
- **Repository**: Generates the data access logic for CRUD operations. It also generates data access logic to handle translations if the entity has translatable attributes.
- **Schema validator:** Generates validation schemas for input validation using Joi (a schema description language and data validator for JavaScript).
- **Controller:** Creates the business logic layer, handling incoming requests and interacting with the repository and other services.
- **Routes:** Defines the API endpoints for the entity, mapping HTTP methods and paths to controller functions while enforcing data validation using the defined schema validator. Listing 2 presents the template used for generating the Express.js router file. During generation, placeholders such as `${controllerName}` and `${entityName}` are substituted with specific entity details based on the entity name, resulting in customized route definitions.
- **Communication:** Generates code for handling asynchronous communication using message queues, facilitating inter-service communication within the microservices architecture. For example, it generates code to subscribe to relevant queues and observe RPC calls, utilizing standard functions provided by the static template skeleton.
- **Tests:** For each entity, it generates two test files: one for unit tests and the other for integration tests.
- **Documentation**: Creates `${entityName}.path.js` and `${entityName}.schema.js` files, handling the entity documentation according to the Swagger specifications.

c After processing all entities, the engine uses other specialized templates to generate common backend infrastructure files, such as:

- **Dockerfile:** For containerizing the microservice.
- **.env files:** Environment configuration files for different stages (development, testing, production).



(a) UI for previewing and configuring the table for the Category entity.



(b) UI for preview and real-time adjustments for the Product entity form.

FIGURE 3. **Overview of the HTML components configuration.**

Each generated backend file is placed into its designated location within the microservices structure established by the static template in Step 2.

d: STEP 4: FRONTEND APPLICATION GENERATION (DYNAMIC TEMPLATES)

Simultaneously, the engine generates frontend components using specialized *dynamic frontend templates*:

Package-level Angular module: Enhancing modularity, a dedicated Angular module `$${packageName}-management.module.ts`) and its associated routing file are generated for the package. This module encapsulates components related to the package's entities, organizing them into entity-specific subfolders to maintain a clear separation of concerns and improve code structure.

UI components: Based on the user configuration for tables and forms, the engine generates specific Angular components via specialized templates. It creates a list component (list-${entityName}.component.html/.ts) using a reusable static table component (dynamic-table.component), as shown in Listing 3. The table headers and actions are dynamic depending on the configuration. It also generates form-based components for manipulating data: add-${entityName}.component, clone-${entityName}.component, and, if translatable fields exist, translate-${entityName}.component.

Core files: For each entity, specialized templates generate essential TypeScript files:

- `${entityName}.model.ts` (data structure),
- `${entityName}.service.ts` (API interaction logic),
- `${entityName}.resolver.ts` (data fetching for routing).

I18n: Collects keys during generation (for enums, labels, messages, breadcrumbs), extracts corresponding values from the generator database for configured languages, and generates JSON files for internationalization.

The generated files are integrated into the frontend application structure established in Step 2, specifically under frontend/src/app/. The Angular module is placed

```
// --- Conceptual Recursive Helper/Partial for
    Rendering Attributes ---
{{#definePartial 'renderAttribute' attribute
    isLastAttribute}}
${attribute.name}: {{#if attribute.isArray}}[{{/if
    }}{
  type: ${attribute.type},
  {{#if attribute.ref}}
    ref: '${attribute.ref}',
  {{else if attribute.subAttributes}}
    {{#each attribute.subAttributes as |subAttr|}}
      {{> renderAttribute subAttr (isLast @index
          attribute.subAttributes)}}
    {{/each}}
  {{/if}}
  {{#each attribute.constraints as |constraint|}}
    {{#unless (isConstraintName constraint.name '
        translatable')}}
      ${constraint.name}: {{formatValue constraint
          .value}}
    {{/unless}}
    {{isNotLast @index attribute.constraints ","}}
  {{/each}}
}{{#if attribute.isArray}}]{{/if}}{{isNotLast
    isLastAttribute ","}}

{{/definePartial}}
// --- Main Template File: model.template.js ---
const mongoose = require("mongoose");
const Schema = mongoose.Schema;
let ${entityName}Schema = new Schema({
  {{#each nonTranslatableAttributes as |attribute
      |}}
  {{> renderAttribute attribute (isLast @index
      nonTranslatableAttributes)}}
  {{/each}}
  {{#if hasTranslatableAttributes}}
  {{#if hasNonTranslatableAttributes}},{{/if}}
    ...
  {{/if}}
}{{#if timestamps === true}}, { timestamps: true
    }{{/if}}
);
...
module.exports = mongoose.model("${entityName}", $
    {entityName}Schema);
```

LISTING 1: Template for generating Mongoose entity schema.

in the components/directory, core files are organized under shared directories (e.g., models are located in shared/models/${packageName}/), and internationalization files are grouped in the assets/i18n/directory.

e: STEP 5: INFRASTRUCTURE & OPERATIONS SETUP (SPECIALIZED TEMPLATES)

The engine uses additional templates to configure the operational environment:

- **Container orchestration**: Generates `docker-compose` files for development and production.
- **Monitoring**: Personalizes static Prometheus and Grafana monitoring templates.
- **CI/CD**: Personalizes static CI/CD pipeline template (.gitbal-ci.yml file).

```
const express = require('express');
const router = express.Router();
const ${controllerName} = require('../controllers/
    ${entityName}.controller');

const validator = require('../validations/${
    entityName}.validator');
${upload ? `const { uploadFile } = require('../
    helpers/helpers');` : ''}
const {validateBody,validateParam,schemas} =
    require('../middlewares/bodyValidation.
    middleware');
router.route('/')
  .get(${controllerName}.getAll)
  .post(${upload}validateBody(validator.schema),${
      controllerName}.add)
  .patch(${controllerName}.updateState);
${hasTranslation ?
`router.route('/translate/:id')
  .get(validateParam(schemas.idSchema, 'id'), ${
      controllerName}.getTranslations)
  .patch([validateParam(schemas.idSchema, 'id'),
      validateBody(validator.translateSchema)],${
      controllerName}.translateData);`
: `` }
...
module.exports = router;
```

LISTING 2: Template for generating Express.js API routes.

```
<div class="sub-component-header pb-3">
  <app-dynamic-table
    [data]="data"
    [cols]="headers"
    [captionConfig]="captionConfig"
    (onDelete)="changeState($event)"
    ${hasTranslation ? `(onTranslateClick)="
        onTranslateClick($event)"` : ``}
    (onEditClick)="onEditClick($event)"
    (onDetailClick)="onDetailClick($event)"
    (onCloneClick)="onCloneClick($event)"
    (onAddClick)="onAddClick()"
    [prohibitDeletion]='${JSON.stringify(
        deleteOneToMany)}'
    [serviceName]="'${camelCase(packageName)}'"
    [componentName]="'${camelCase(entityName)}'">
  </app-dynamic-table>
</div>
```

LISTING 3. Template for generating angular list component (HTML table).

f: STEP 6: SOURCE CODE SUBMISSION AND ADAPTATION

Finally, the engine organizes all generated and personalized files. The complete source code is automatically pushed to a designated Git repository, notifying the adaptation team. The package enters the testing and adaptation phase for further customization, testing, and integration.

B. TECHNICAL ARCHITECTURE

This section details the technical foundation and operational environment of FSMicroGenerator, covering the technologies employed and the secure infrastructure designed to support its functions.

1) CORE TECHNOLOGIES AND COMPONENTS

FSMicroGenerator utilizes a modern technology stack for robustness and flexibility. The backend is built with Node.js

and Express.js, providing a REST API for data handling, while MongoDB stores project configurations and metadata. The UI employs Angular for a responsive experience, supported by a translation system for accessibility. Swagger provides clear API documentation, facilitating interaction with the tool's various services. The entire system is containerized using Docker for portability and consistent deployment across environments.

2) SECURE INFRASTRUCTURE AND OPERATIONAL MODEL

Although it can operate independently, FSMicroGenerator infrastructure design was inspired by DevSecOps principles, with the objective of providing an ecosystem that leverages automation and collaboration while securing assets. This secure and integrated ecosystem comprises FSMicroGenerator server, a private internally hosted GitLab instance, and a dedicated development server, all operating within a protected local network. This isolation is fundamental to the security approach, preventing external exposure of source code and company assets.

Within this controlled environment, FSMicroGenerator securely integrates with the private GitLab instance. Generated code is automatically pushed to the appropriate repository, ensuring secure storage and centralized version control. Access to GitLab is strictly restricted, primarily allowing interaction from the tool's server and the development server. For adaptation and testing, the adaptation team clones the generated project to the development server upon request, enabling internal validation, customization, and refinement within the secure zone. This process reinforces CI/CD practices, with changes tracked via Git and validated through automated pipelines, all within the protected ecosystem.

Security is woven throughout the ecosystem's design and operation. Developers interact with the system primarily through FSMicroGenerator's controlled graphical interface, limiting direct exposure to core components. Strict role-based access control (RBAC) is enforced, and GitLab accounts are automatically synchronized based on user status within the managed environment, ensuring permissions remain current and appropriate. All critical actions are logged for traceability and audit purposes. Furthermore, sensitive data, such as configurations and API keys, are encrypted at rest, while secure communication protocols protect data exchanges between the frontend and backend, as well as interactions with GitLab and the development server via secure API keys. This holistic security approach ensures data protection, access control, and the integrity of generated projects throughout their lifecycle within the ecosystem.

IV. ARCHITECTURE OF THE GENERATED SOLUTIONS

This section presents the overall architecture of the applications generated by FSMicroGenerator. It covers the functional structure, the applicative layers, and the underlying technical foundations. Figure 4 offers a high-level view of the generated solutions architecure, summarizing the main components and their interactions. It serves as a reference for the detailed descriptions that follow.

A. FUNCTIONAL ARCHITECTURE

Generated solutions feature a modular functional architecture designed to meet business requirements by automating essential tasks, allowing developers to focus on strategic aspects. Figure 5 illustrates the main features integrated into the generated solutions. Core to many applications are CRUD operations, which form the foundation for data management. However, manually implementing these essential operations can be repetitive and time-consuming, often leading to inconsistencies due to varying coding styles and posing maintenance challenges. FSMicroGenerator addresses this by automating the generation of standardized CRUD operations, ensuring structured, homogeneous code that adheres to data validation standards, thus minimizing errors and improving maintainability. Beyond CRUD, advanced capabilities encompass entity cloning for variant management, centralized multilingual support for both dynamic entities and static UI elements, and sophisticated search/filtering functionalities. To further reduce developer workload and ensure consistency, pre-configured services are integrated, such as authentication, RBAC, and dynamic menu management. Furthermore, the solutions include tools for dynamic dashboard generation, reporting, and data import/export, facilitating decision-making and system integration.

B. APPLICATIVE ARCHITECTURE

1) LAYERED ORGANIZATION

The architecture is designed around a multi-layered organization, ensuring a separation of concerns and facilitating the maintainability and scalability of the generated solutions.

- **Presentation layer:** This layer defines the user interface, designed for efficient and ergonomic navigation while integrating essential elements of the business process management. The goal is a smooth and intuitive user experience, achieved by reducing cognitive load. Uniform visual and functional behavior ensures this by providing a predictable and coherent interface. Logical grouping of information is employed to enhance readability, while instant visual feedback confirms user actions to reduce uncertainty. Optimized response times and smooth rendering contribute to seamless and efficient interactions. Furthermore, its decoupled connection via the API Gateway allows the interface to evolve independently without impacting backend services.
- **API Gateway layer**: This layer plays a key role in request management and service orchestration. It performs several essential functions, including: I) Service registry for dynamic microservice management; II) Enforcement of security policies via centralized access. III) Routing requests to appropriate microservices based on endpoints; IV) Load balancing for

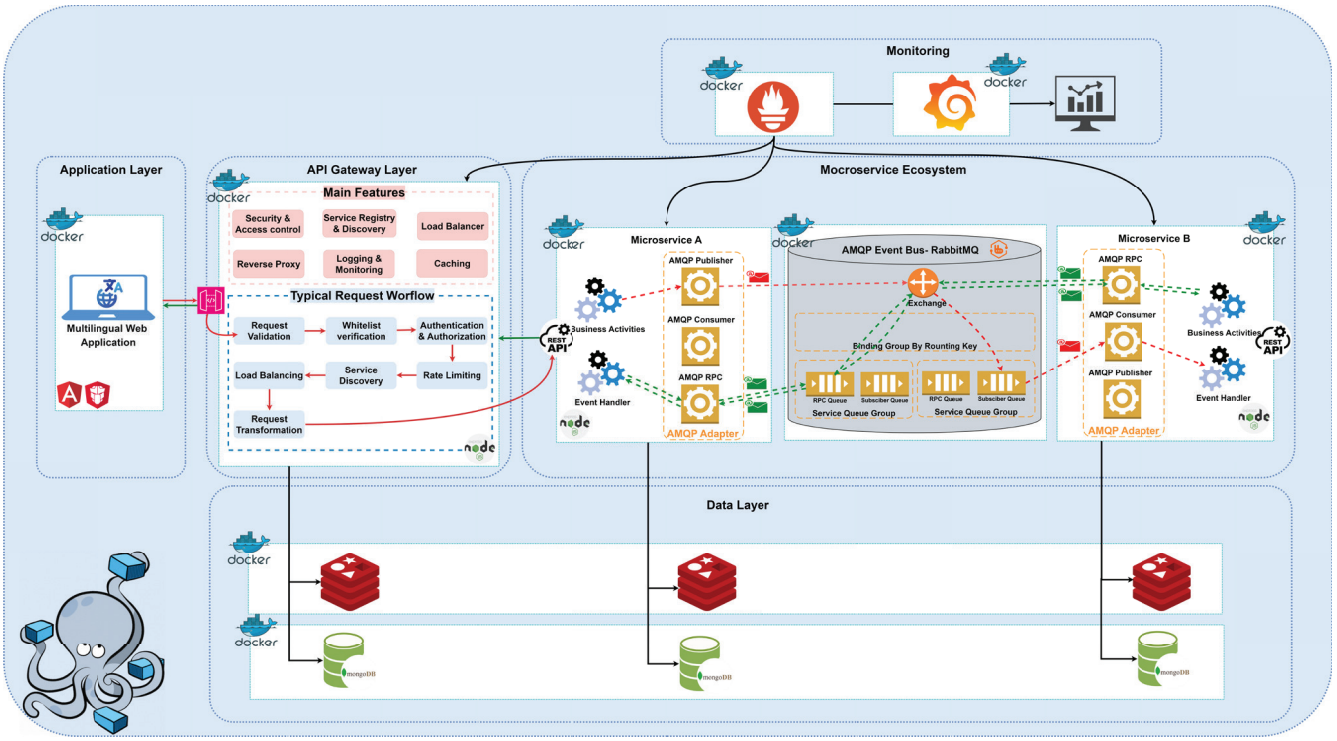


FIGURE 4. High-level architecture of solutions generated by FSMicroGenerator, highlighting key components and their interactions.

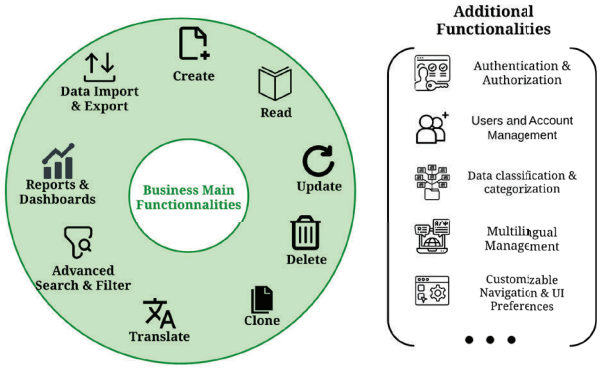


FIGURE 5. Overview of core and some additional functionalities in the solutions generated by FSMicroGenerator.

balanced request distribution and optimized performance; This simplifies backend access while ensuring scalability and controlled interactions.

- **Microservices layer:** This layer consists of independent, specialized services that expose their functionality via REST APIs. For each generated package, there is a main microservice for core business logic, supplemented by essential support services. These include Member management (for users, roles, permissions via RBAC, and authentication), Technical configuration (handling dynamic menus, and internationalization), and Taxonomy (for structuring and classifying content). Each microservice operates autonomously, allowing for independent updates, which ensures optimal modularity, scalability, and maintainability for the generated solutions.
- **Data layer:** This layer is responsible for data storage, management, accessibility, consistency, and integrity. It implements a database per service pattern, isolating data for each microservice to enhance scalability and resilience by preventing data access conflicts and limiting failure impact. Performance is optimized using a Redis caching system, temporarily storing frequently accessed data to reduce query latency, improve responsiveness, and reduce the load on storage services.

2) SECURITY AND ACCESS GOVERNANCE

Reflecting the importance placed on security within FSMicroGenerator's own infrastructure, security is also treated as a central pillar of the solutions it generates. This is achieved through robust measures. The API Gateway acts as the sole, secure entry point, managing authentication and authorization via JSON Web Token, preventing direct access to internal services. Generated solutions integrate data encryption (especially for sensitive data), robust protection against common web vulnerabilities (injection, Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF)) through double frontend and backend data validation, and rate limiting against brute-force and Distributed Denial of Service (DDoS) attacks. Secure inter-service communication protocols and managed frontend authentication tokens (preventing XSS

exposure) are employed. Access governance is maintained through a strict RBAC system, ensuring compliance and a protected user experience.

3) SCALABILITY AND MAINTENANCE

The application architecture of the generated solutions is built for optimal scalability. Applications can handle growing workloads without compromising performance. Horizontal scaling enables microservices to be replicated and balanced dynamically, ensuring efficient traffic management and high availability. The API Gateway supports this flexibility by orchestrating services and allowing new microservice instances to be added without disrupting the system.

Performance is improved through Redis caching, which enhances application responsiveness. Docker containerization ensures scalable and reproducible deployments, making infrastructure expansion easier. Automated API documentation using Swagger simplifies integration and improves developer onboarding. The modular design supports maintainability by enabling updates without regression risk, ensuring smooth and continuous scalability.

C. TECHNICAL ARCHITECTURE

FSMicroGenerator uses proven technologies to deliver modular, scalable, and maintainable solutions. It reduces technical barriers and simplifies complex architectures, helping teams work more efficiently despite the shortage of specialized developers. This allows them to focus on business needs rather than struggling with complex technical challenges.

1) MICROSERVICES: A DISTRIBUTED AND ADAPTIVE ARCHITECTURE FOR MODERN APPLICATIONS

The architecture of a web application plays a fundamental role in its performance, scalability, and maintainability. Like the skeleton of a building, it provides structure and determines how different parts interact. A well-designed architecture supports change and simplifies communication between components.Two common architectural approaches are the monolithic and microservices [22].

The monolithic architecture bundles all components into one unit. This simplifies early development but creates scaling and maintenance issues as the system grows. In contrast, microservices split the application into independent services, improving modularity and fault isolation [23]. This enables faster deployment and adaptability to changing needs, which many companies value for shorter time-to-market [24]. However, microservices introduce complexity, especially for new developers, due to the need for expertise in distributed systems management, service orchestration, and inter-service communication [24]. FSMicroGenerator directly addresses these complexities by automating service generation, inter-service communication, and infrastructure setup. This reduces technical overhead and allows developers to focus on business logic.

A key challenge in microservices is communication: since services operate independently, they must exchange data reliably to maintain consistency, integrity, and coherence across the system. Two main communication models are commonly used:

- Synchronous communication follows the request-response pattern using the remote procedure call (RPC) model. In this model, a service sends a request and waits for a direct response before continuing execution [25]. While this is well-suited for real-time interactions, it increases coupling between services and is sensitive to latency and failures, which can delay or block execution [26].
- Asynchronous communication relies on an event-driven model, where services publish and consume messages via a message broker such as RabbitMQ or Kafka [25]. Unlike synchronous methods, the sender does not wait for a response and continues execution immediately. This improves decoupling, scalability, and fault tolerance but adds complexity in managing message delivery, ordering, and retries [26].

FSMicroGenerator adopts a hybrid approach, using RabbitMQ, combining synchronous RPC for time-sensitive interactions and asynchronous messaging for background and decoupled tasks. This approach balances responsiveness, resilience, and simplicity.

The choice of RabbitMQ over Kafka was driven by several factors. Kafka is optimized for large-scale, high-throughput event streaming and scenarios requiring durable event storage and high fault tolerance. However, Kafka requires a more complex infrastructure and presents a steeper learning curve. In contrast, RabbitMQ is easier to deploy and manage, making it more suitable for transactional communication, low-latency systems, and applications that benefit from advanced queue management [25]. Its smoother integration with microservices and simpler setup make RabbitMQ a better fit for FSMicroGenerator's goal of combining robustness with ease of use.

2) MEAN STACK: A COHERENT ECOSYSTEM FOR UNIFIED DEVELOPMENT

In web development, the use of technology stacks has become essential for efficiently structuring application design. A stack refers to a coherent set of technologies used to build a complete application, from the frontend to the database [27]. Companies increasingly seek full-stack developers who can manage both user interfaces and backend services. According to the Stack Overflow 2024 survey [28], 30.7% of developers identify as full-stack. It is the most reported role for the third year in a row, indicating a shift toward multi-skilled development positions.

Web applications are built using combinations of languages, frameworks, and libraries rather than a single technology. Common stacks such as L/WAMP (Linux/Windows, Apache, MySQL, PHP/Python/Perl) or the.NET stack have

been widely used over the years. However, they combine heterogeneous tools and languages, each requiring specific expertise, often exceeding the skill set of a single developer [30]. This fragmentation increases development costs, team size, and coordination complexity. Additionally, these technologies were not originally designed to work together, making integration and maintenance more difficult [30]. To address these limitations, more homogeneous alternatives like the MEAN stack (MongoDB, Express.js, Angular, Node.js) have emerged as modern, efficient solutions, especially suited to microservices architectures.

The MEAN stack's main strength lies in its consistency: JavaScript is used across all layers, which simplifies integration, reduces context-switching, and improves maintainability. As JavaScript has been the most widely used language for over a decade [28], MEAN benefits from a large support ecosystem, ensuring stability and long-term viability. Its modular structure allows components to evolve independently. MongoDB offers flexible NoSQL data storage; Node.js supports asynchronous operations; Express.js simplifies API routing; and Angular delivers responsive, client-side interfaces. MEAN also enhances performance by using JSON end-to-end, eliminating the need for data format conversions. MongoDB stores data in BSON, reducing query time and server load. Additionally, MEAN aligns well with agile methods and short development cycles, supporting rapid prototyping (e.g., minimum viable product). Its open-source model removes licensing costs and vendor lock-in, making it scalable and cost-effective.

By adopting the MEAN stack, FSMicroGenerator delivers a unified, modular, and high-performance development framework that enables faster delivery, improved scalability, and better interoperability for modern web applications.

3) DEVELOPER EXPERIENCE AND DOCUMENTATION WITH SWAGGER AND PRIMENG

With an ongoing focus on minimizing technical overhead and allowing teams to concentrate on business logic, FSMicroGenerator prioritizes developer experience. It integrates tools that simplify service design, streamline interaction with microservices, and reduce the effort required for configuring UIs.

On the backend, Swagger automates API documentation and provides an interactive interface for exploring endpoints. This removes the need for manual specification writing, reduces miscommunication between frontend and backend teams, and accelerates API integration.

On the frontend, PrimeNG, an Angular component library, simplifies the development of responsive UIs. It provides a wide range of pre-built UI elements, enabling fast, consistent, and customizable interface creation. This ensures visual and ergonomic consistency across generated applications while allowing developers to tailor interfaces to specific business needs.

This combination supports a structured and efficient development process, enabling faster delivery and better collaboration across teams.

4) DOCKERIZED APPLICATIONS: A RELIABLE APPROACH FOR MODULAR DEPLOYMENTS

Deploying the applications generated by FSMicroGenerator involves coordinating multiple components—frontend, API Gateway, and microservices—each with its own dependencies and runtime requirements. Managing these across development, testing, and production environments can introduce inconsistencies, errors, and manual configuration overhead.

To address this, FSMicroGenerator adopts Docker, a containerization technology that packages each component along with its dependencies into isolated environments [31]. Unlike virtual machines, which require a full operating system per instance, Docker containers share the host system's kernel. This makes them lighter, faster, and more efficient [29]. According to the Stack Overflow 2024 survey, 58.7% of professional developers use Docker to deploy distributed systems, highlighting its adoption as a standard tool for modular deployment [28].

For each generated component, FSMicroGenerator automatically produces a Dockerfile and a.dockerignore. The Dockerfile defines the image build process, including base images and dependencies, using lightweight base images to optimize performance. The.dockerignore file reduces image size by excluding unnecessary files. To orchestrate the full system, FSMicroGenerator automatically generates two Docker Compose files: one for development and one for production. Each file defines the full application stack, including the frontend, API Gateway, and microservices, along with required infrastructure services such as MongoDB, Redis, RabbitMQ, Prometheus, and Grafana. This setup allows developers to launch consistent and isolated environments with a single command, tailored to each stage of the lifecycle.

By integrating Docker and Docker Compose, FSMicroGenerator enables fast, reproducible, and modular deployments. Developers can focus on functionality rather than environment setup, reducing configuration errors and supporting scalable, container-based application delivery.

5) CI/CD AND DEVSECOPS-DRIVEN WORKFLOW FOR SCALABLE APPLICATIONS

To support fast, reliable, and secure delivery of generated solutions, FSMicroGenerator integrates DevOps and DevSecOps practices into a GitLab-based continuous integration and continuous deployment (CI/CD) pipeline (see Figure 6). CI/CD automates all stages of the delivery process—from code integration and testing to security validation and deployment—reducing manual work, improving consistency, and accelerating release cycles.

The integrated.gitlab-ci.yml file defines a structured pipeline with seven stages: install, test, build, security, deploy,

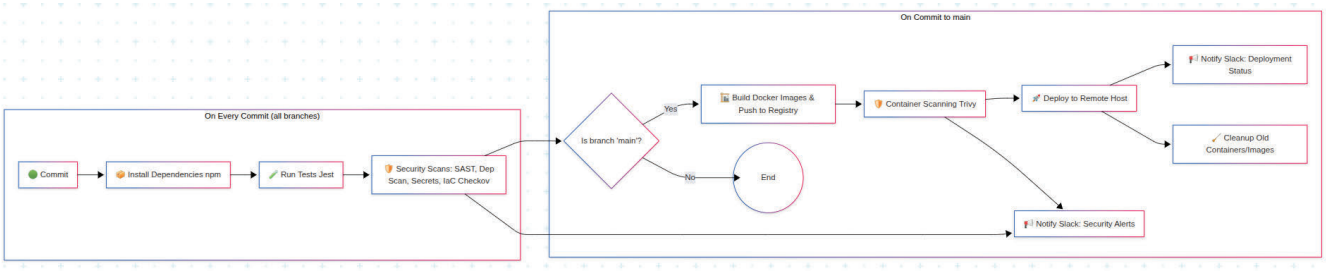


FIGURE 6. CI/CD pipeline embedded in FSMicroGenerator-generated applications, automating test, security, build, deployment, and cleanup using DevSecOps practices.

notify, and cleanup. Each stage manages the application components—frontend, API Gateway, and microservices—independently and reliably. Dependencies are installed, tests are executed, containers are built and tagged, and deployments are packaged and executed without manual intervention. By default, container images are pushed to the GitLab Container Registry.

Security is integrated as a native part of the pipeline, not treated as a post-deployment step. FSMicroGenerator applies DevSecOps principles by embedding automated security checks: static code analysis (SAST), dependency scanning, secret detection, and container image scanning. While most checks run during the CI process, container image scanning is performed during the CD stage. An additional infrastructure-as-code (IaC) scan using Checkov detects misconfigurations early. If high-severity issues are found, Slack alerts are automatically triggered. This shift-left approach reduces the risk of exposing vulnerabilities in production.

Deployment to production is managed via a remote SSH step, where the appropriate Docker Compose file is pulled and executed on the target environment. Each component runs in its own container, with image versions tagged by the Git commit SHA. If deployment fails, a rollback mechanism restores the last stable version. Once deployment is complete, the cleanup stage removes unused containers, dangling images, and orphaned resources to maintain a stable and efficient environment.

To close the loop, automated notifications provide real-time feedback on pipeline health. Slack messages report deployment success or failure and highlight detected vulnerabilities, improving team responsiveness and transparency.

Through this CI/CD and DevSecOps-driven workflow, FSMicroGenerator ensures that generated applications are functional, scalable, security-hardened, and production-ready—with minimal operational overhead.

6) PROMETHEUS AND GRAFANA: PROACTIVE MONITORING FOR HIGH-PERFORMANCE APPLICATIONS

In distributed and scalable architectures, monitoring is a critical function that ensures system performance and stability, especially in production environments [24], [32]. Without robust observability, it becomes difficult to detect bottlenecks, trace failures, or optimize resource usage. To address this, FSMicroGenerator integrates Prometheus and Grafana, an open-source monitoring stack that enables complete, real-time visibility across all generated services.

FSMicroGenerator adopts a generic and automated monitoring approach. Each generated microservice includes built-in monitoring instrumentation and exposes a/metrics endpoint in a format compatible with Prometheus. The API Gateway, leveraging its service registry capability, provides Prometheus with a dynamic list of registered microservice endpoints to scrape. This automated setup eliminates the need for manual configuration, ensuring consistent monitoring coverage and simplifying the integration of newly generated services into the observability layer. As a result, monitoring becomes a default feature of every microservice, increasing system transparency and maintainability without requiring developer intervention.

Beyond basic anomaly detection, FSMicroGenerator supports predictive maintenance by enabling the continuous collection and analysis of system metrics. Metrics such as memory consumption, CPU usage, response time, and HTTP error rates are tracked in real time. A gradual increase in latency, rising memory usage, or a spike in HTTP 500 errors can reveal latent issues before they cause visible outages [33]. By acting on these early signals, teams can optimize resource allocation, refactor code, or scale specific services to prevent performance degradation. FSMicroGenerator ensures this by automatically injecting Prometheus-compatible measurement points into every generated service, allowing for fine-grained monitoring with minimal effort.

To support rapid deployment of monitoring infrastructure, FSMicroGenerator provides a ready-to-use configuration. This includes a prefilled prometheus.yml file defining scrape targets, a default Grafana data source, and standard dashboard templates. Alerting is also preconfigured with default notification channels (including Slack) and predefined alert rules for common incidents such as high memory usage or service unavailability. These defaults accelerate the setup process and ensure operational readiness from the first deployment.

The choice of Prometheus and Grafana is based on performance, flexibility, and open standards. Prometheus, a widely adopted solution in modern cloud-native environments, uses a pull-based model that allows it to periodically collect metrics from services without requiring agents [34]. Its time-series

database structure efficiently stores temporal data, enabling precise trend analysis, anomaly detection, and long-term performance tracking. In contrast to proprietary tools like Datadog or New Relic, Prometheus is open-source and adaptable, with no licensing restrictions. Grafana complements Prometheus by offering an intuitive interface for visualizing metrics, building dashboards, and interacting with time-series data. Unlike Kibana, which is better suited for log analysis, Grafana focuses on real-time performance metrics, providing more relevant insights for microservice monitoring and operations.

By embedding this monitoring stack, FSMicroGenerator delivers a comprehensive observability solution. It empowers teams to make data-driven decisions, respond quickly to failures, and continuously improve system performance. This proactive approach to monitoring improves service availability, supports predictive diagnostics, and enhances the overall resilience of generated applications in production environments.

7) STANDARDS AND BEST PRACTICES AT THE HEART OF GENERATED SOLUTIONS

FSMicroGenerator integrates recognized development standards to ensure that generated solutions are robust, consistent, and maintainable. At design time, it adopts Object Management Group (OMG) standards by using UML class models to define business entities. This model-driven approach ensures that the conceptual structure aligns with the generated software architecture. The tool also applies RESTful API design best practices, enforcing proper use of HTTP methods, consistent response formatting, and clear error handling, which improves interoperability with external systems. Figure 7 presents the key principles and best practices applied in designing RESTful microservices APIs.

Security and software quality are addressed from the start. Following a DevSecOps approach, FSMicroGenerator embeds automated security checks and continuous testing into the development lifecycle. This integration helps identify vulnerabilities early and ensures that security is enforced without requiring manual intervention. Architectural design is based on proven patterns, both in terms of microservices architecture (e.g., API Gateway, Database per Service, Event-Driven Architecture) and software development. It incorporates design patterns such as the repository pattern to structure data access, the factory pattern to promote dynamic instance creation, and the singleton pattern to optimize management of critical services (e.g., Mongodb, Redis, RabbitMQ). These patterns improve modularity, scalability, and reduce coupling between components.

Frontend consistency is ensured through the use of standardized Angular components, particularly those provided by PrimeNG. FSMicroGenerator also promotes the reuse of shared logic across both backend and frontend, reducing redundancy and easing maintenance. This combination of model-driven design, security integration, and standardized architecture enables the rapid generation of reliable, scalable, and production-ready microservice solutions.

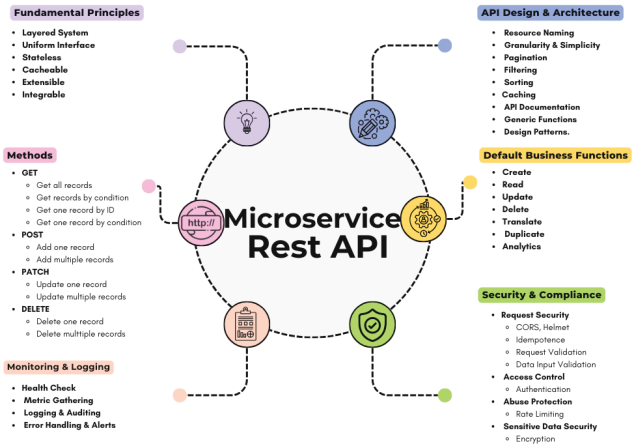


FIGURE 7. Key principles and best practices driving our approach to designing RESTful microservices APIs.

V. METHODOLOGY

This section presents the experimental protocol used to evaluate FSMicroGenerator. The goal is to compare its impact on development outcomes with a traditional manual approach. We outline the case study, the development conditions, and the evaluation metrics. We also explain how variations in implementation completeness were handled. Finally, we discuss threats to the validity of the study.

A. CASE STUDY DESCRIPTION

To provide a realistic comparison context, we selected an inventory management system as the experimental case study. Although simplified for controlled evaluation, this system was designed to encompass typical functionalities of enterprise web applications. Its domain model was formalized using a UML class diagram structured according to domain-driven design principles. The system was decomposed into distinct bounded contexts, each intended to be implemented as an independent microservice.

As shown in Figure 8, the architecture differentiates between core business modules (white-labeled packages) and reusable technical subsystems (green-labeled packages) intended for cross-module integration. These reusable subsystems were pre-generated by FSMicroGenerator for its implementation path and provided as pre-built components. This ensured that the focus remained on developing core business logic and integration, rather than rebuilding foundational utilities from scratch in both scenarios. The evaluation focused on how effectively each approach facilitated the development and integration of the core business modules while leveraging these shared components and maintaining architectural integrity. This full-stack case study encompasses backend, frontend, and infrastructure considerations,

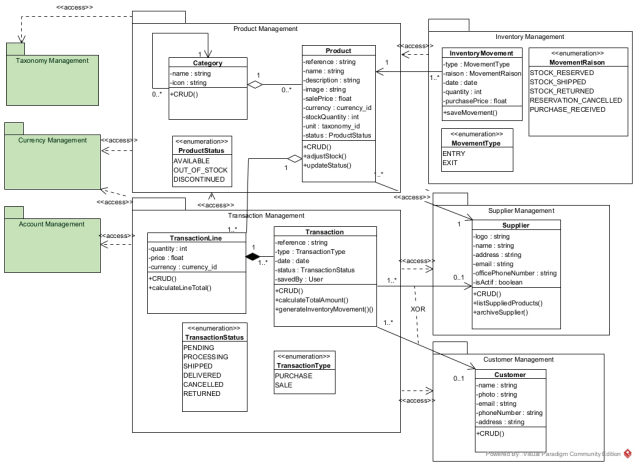


FIGURE 8. UML class diagram of the inventory management case study used in the experimental evaluation.

allowing for assessment across functional and non-functional dimensions within a manageable scope.

B. EXPERIMENTAL SETUP AND DEVELOPMENT PROTOCOLS

The experiment involved two parallel development efforts under identical environmental conditions: the same functional and non-functional target scope (detailed in Appendix A, Table 5 and Appendix B, Table 6, respectively), the same server infrastructure, and the same underlying technology stack (MEAN stack). A strict maximum development time of 8 hours was allocated to both efforts to simulate time-constrained project conditions.

A key aspect of this study involves the composition of the development teams, which was intentionally asymmetric. The manual implementation was performed by a team of five developers, each possessing 2-3 years of professional experience with the MEAN stack and microservices. The FSMicroGenerator-based implementation was undertaken by a team of two junior developers with 1 and 2 years of experience, respectively. This asymmetry was chosen to explore the tool's potential impact under varied team conditions, but it introduces factors beyond the tool itself, as discussed in the threats to validity (Section V-E).

C. HANDLING INCOMPLETE IMPLEMENTATIONS

A significant difference emerged in the scope of implementation completed within the 8-hour timeframe. The FSMicroGenerator implementation achieved full coverage (100%) of both specified functional and non-functional requirements, while the manual implementation achieved partial coverage (61% functional, 33% non-functional, as detailed in Appendix A, Table 5 and Appendix B, Table 6). Directly comparing raw metrics related to effort, codebase size, or performance would be misleading under these circumstances.

To enable a more meaningful comparison, we introduced completion-aware normalization using two ratios derived from the checklists in the appendices:

- $\mathbf{R}_f$: Functional coverage ratio, defined as the number of functional features implemented relative to the total expected (Equation 1).
- $\mathbf{R}_{nfr}$: Non-functional coverage ratio, defined as the number of non-functional requirements (NFRs) implemented relative to the total, accounting for both system-wide and module-specific NFRs (Equation 2).

$$\mathbf{R}_f = \frac{\text{Functional features implemented}}{\text{Total functional features defined}} \tag{1}$$

$$\mathbf{R}_{nfr} = \frac{\text{Total NFR coverage achieved}}{\text{Total NFRs expected}} \tag{2}$$

For metrics where higher values typically correlate with increased scope (e.g., lines of code, resource usage, manual interventions), the raw values from the partially completed manual implementation were adjusted upwards using Equation 3. Conversely, for metrics like throughput, where increased complexity and scope might negatively impact performance, the raw value was adjusted downwards using Equation 4

$$\text{Normalized value} = \frac{\text{Raw value}}{R_f \times R_{nfr}} = \frac{\text{Raw value}}{0.2013} \tag{3}$$

$$\text{Adjusted throughput} = \text{Raw throughput} \times R_f \times R_{nfr} \tag{4}$$

This normalization provides an estimate of the values that could have been observed if the manual implementation had reached full scope. However, this approach relies on assumptions about the scalability of metrics relative to scope. The limitations and potential impact of these assumptions on the study's conclusions are discussed in Section V-E.

D. EVALUATION CRITERIA AND DATA COLLECTION

A comprehensive set of quantitative and qualitative metrics was defined to compare the two implementations, as summarized in Table 1.

Quantitative metrics capture objectively measurable attributes (e.g., code volume, system performance, time, automation). Data was collected using standardized tools such as Git logs, Apache JMeter, and Docker stats. Normalization was applied to relevant quantitative metrics for the manual implementation.

Qualitative metrics assessed architectural and structural properties requiring expert judgment, including modularity, clarity, reusability, and maintainability. These were evaluated through a structured manual review conducted independently by two evaluators using a predefined rubric (detailed in Appendix C, Table 7) based on a 5-point Likert scale. To ensure consistency and minimize bias, discrepancies greater than one point between reviewers were resolved through discussion. Qualitative metrics were not normalized, as the overall architectural structure was present in both implementations, allowing for assessment even with differing internal feature completeness.

TABLE 1. Evaluation metrics used to compare FSMicroGenerator and manual development, including objectives, data collection tools, and normalization ratios.

Category	Metric	Objective	Data collection tool	Normalized by
Quantitative	Functional coverage ratio (R_f)	Quantify the proportion of functional features implemented relative to the total required	Manual feature checklist	—
	Non-functional coverage ratio (R_{nfr})	Quantify the proportion of non-functional requirements implemented relative to the full specification	Manual NFR checklist	—
	Development time	Measure productivity under a fixed time constraint	Tracked via stopwatch	—
	Manual interventions	Assess the degree of automation by counting manually written and modified files	Analyzed via commit logs and diff inspection	R_f, R_{nfr}
	Lines of code (LOC)	Estimate codebase size and complexity	VS Code Counter extension	R_f, R_{nfr}
	Scalability under load	Evaluate system performance as the number of users increases	Apache JMeter (latency, error rate, throughput)	R_f, R_{nfr}
	Resource usage (CPU/RAM)	Measure runtime efficiency under simulated load	Docker stats	R_f, R_{nfr}
Qualitative	Architectural modularity	Assess the separation of concerns, cohesion, and service decomposition	Expert review (rubric-based)	
	Code reusability	Evaluate the presence of reusable components and abstraction mechanisms		
	Code clarity, structural consistency, and standard conformance	Assess uniformity of structure across modules, adherence to naming conventions and architecture standards (MVC, RESTful)		
	Maintainability	Estimate modifiability, independence of components, and technical debt risk		—

The metrics were selected in accordance with ISO/IEC 25010:2023 and ISO/IEC 25002:2024, ensuring coverage of relevant quality characteristics and adherence to standard measurement practices. Table 1 provides a summary of these metrics, their objectives, data collection methods, and the normalization strategy applied.

E. THREATS TO VALIDITY

To ensure transparency and guide interpretation, this subsection outlines considerations regarding the validity of our findings, reflecting standard practices in empirical software engineering research.

1) CONSTRUCT VALIDITY

Construct validity concerns the extent to which our operational measures accurately reflect the theoretical constructs being investigated (i.e., development effort, productivity, code quality).

- **Team composition:** The study design involved comparing a team of two junior developers (1 and 2 years of experience) using FSMicroGenerator with a team of five developers (2-3 years experience) performing manual implementation. This asymmetry, chosen to explore the tool's potential under varied conditions, means that factors such as team size, experience levels, and potential coordination differences should be considered when interpreting the results. The observed outcomes reflect the interplay between the development approach and the specific team context, positioning this study as an exploratory comparison rather than a direct measure of the tool's isolated effect under identical team conditions.
- **Normalization approach:** To compare implementations with differing completion levels, we employed normalization based on functional and non-functional coverage. This approach assumes a generally proportional relationship between scope and metrics like effort or code size. As complexity can scale non-linearly in software development, the normalized values for the manual implementation serve as estimates to facilitate comparison, rather than precise predictions of outcomes at full scope. This estimation aspect introduces a degree of uncertainty for constructs involving normalization.

2) EXTERNAL VALIDITY CONSIDERATIONS (GENERALIZABILITY)

External validity addresses the extent to which findings might apply in different settings.

- **Context specificity:** The findings are based on an inventory management system case study. While this represents a common type of complex web application, the specific results may vary when applied to significantly different application domains or project scales. However, the underlying principles of code generation for structure and boilerplate are expected to offer benefits across a wide range of web application contexts.
- **Participant characteristics:** The study involved specific developers (two junior, five with 2-3 years of experience). Outcomes might vary with developers possessing different skill sets, experience levels, or working within different team dynamics.
- **Tool specificity:** Findings are specific to FSMicroGenerator. While potentially indicative of broader trends in code generation, direct extrapolation to other tools with

different features or design philosophies may not be appropriate.

3) CONCLUSION VALIDITY CONSIDERATIONS

Conclusion validity concerns the reasonableness of the conclusions drawn from the data.

- **Basis of comparison:** The comparison relies on data from single instances (one generator project, one manual project) and qualitative assessments. While informative, the absence of multiple replications suggests caution when drawing broad statistical conclusions from these specific observations.
- **Interpretation of estimates:** As discussed, the interpretation of normalized data requires acknowledging the underlying assumptions. Conclusions based on these estimates should be presented with appropriate consideration for their estimation basis.

Acknowledging these considerations provides a transparent account of the study's context and boundaries, encouraging careful interpretation of the results.

VI. RESULTS AND ANALYSIS

This section presents the comparative evaluation results for the FSMicroGenerator-based implementation and the manual implementation. We analyze both quantitative metrics, including normalized values to account for differing completion levels, and qualitative assessments based on expert review. The analysis aims to interpret the observed differences within the context described in Section V.

A. QUANTITATIVE ANALYSIS

The two development efforts resulted in significantly different levels of completion within the 8-hour time limit: 100% functional and non-functional coverage for the FSMicroGenerator-based implementation versus 61% functional and 33% non-functional coverage for the manual approach. This disparity necessitates the use of normalization (based on $\mathbf{R}_f = 0.61$, $\mathbf{R}_{nfr} = 0.33$) for metrics influenced by scope, allowing for a tentative comparison. Table 2 presents both the raw measurements and the normalized estimates for the manual implementation alongside the results for the FSMicroGenerator implementation.

1) DEVELOPMENT EFFORT

Both efforts operated under the same 8-hour time constraint. The FSMicroGenerator implementation yielded a fully functional system within 3 hours, requiring only 11 commits and 148 manual file modifications post-generation, primarily for UI adjustments and features outside the tool's scope (see Appendix A). In contrast, the manual team utilized the full 8 hours, achieving only partial completion. Their effort involved 115 commits, the creation of 3,835 files, and modification of 1,305 files.

Applying the normalization factor (dividing by 0.2013) to estimate the effort for full completion yields figures of approximately 40 hours, 571 commits, and over 25,000 manual file interventions for the manual approach. While these normalized figures are extrapolations based on the assumption of proportional effort scaling (see Section V-E), they suggest a substantial difference in the projected effort required to reach full scope. FSMicroGenerator demonstrably automated significant portions of the setup, configuration, and boilerplate code across the full stack, allowing developers to focus on integrating specific business logic and achieving full coverage rapidly. The manual team, despite having more cumulative experience, expended considerable effort on foundational work, reflected in the high number of file operations even for partial completion.

2) CODEBASE SIZE

The FSMicroGenerator implementation resulted in a larger final codebase (904 files, ∼192k LOC) compared to the raw output of the manual implementation (176 files, ∼26k LOC). However, the manual team created over 3,800 files during development, suggesting significant code churn, refactoring, or deletion, potentially indicative of challenges in establishing a stable structure under time pressure. Normalizing the final manual codebase size yields estimates of 874 files and ∼130k LOC.

These figures suggest that a fully completed manual implementation might approach the size of the generator-based one. The FSMicroGenerator-based implementation's larger size is partly due to the automatic inclusion of features beyond the core requirements, such as entity cloning, business statistics visualization components, and database-structured documents for pre-configured entities (e.g., menus, users, roles), aimed at greater production-readiness. Therefore, direct comparison of LOC or file count as a measure of efficiency is complex; the generator produces more extensive code automatically, while the manual process involved considerable exploratory coding reflected in the high number of created files versus the final count.

3) PERFORMANCE UNDER LOAD

To assess system scalability and performance under load, we executed a load test using Apache JMeter. The scenario involved 250 concurrent virtual users, each performing two iterations of a sequence consisting of three operations: a POST request to create a product (with randomized attributes), a GET request to retrieve all products (to simulate read load under dynamic data conditions), and a DELETE request to remove the product created in the same iteration. A 60-second ramp-up period was used to gradually apply the load. This resulted in a total of 1500 requests per implementation. Latency, throughput, and error rates were recorded during execution.

This test revealed notable performance trade-offs. The manual implementation showed significantly lower raw latency (15.21 ms) compared to the FSMicroGenerator implementation (1,828.38 ms). However, this raw speed corresponds to a system with incomplete functionality

TABLE 2. Quantitative evaluation: raw and normalized results.

	Metric	Manual (raw)	Manual (normalized)	FSMicroGenerator
Coverage ratios	Functional coverage (R_f)	0.61	—	1.00
	Non-functional coverage (R_{nfr})	0.33	—	1.00
Development effort	Development time (hours)	8.00	39.74	3.00
	Number of commits	115	571	11
	Files created manually	3,835	19,051	0
	Files modified manually	1,305	6,482	148
Codebase size	Number of files	176	874	904
	Lines of Code (LOC)	26,209	130,198	192,050
Scalability under Load	Latency (ms)	15.21	75.56	1,828.38
	Throughput (req/sec)	25.44	5.12	21.75
	Error rate (%)	0.00	0.00	0.00
Resource usage (under load)	CPU usage (%)	19.24	95.58	136.12
	RAM usage (MB)	641.99	3,189.20	1,291.80

and fewer processing layers (e.g., missing security, data validation, caching, and inter-service communication). The higher latency of the FSMicroGenerator implementation reflects the overhead of its more complete feature set, including middleware execution (authentication, rate limiting, data validation), as well as deeper service chains and inter-service data interactions required by the full scope.

Throughput tells a related story. The raw throughput of the manual system was higher (25.44 req/sec) than the generator's (21.75 req/sec). However, applying the normalization factor (multiplying by 0.2013) to adjust for the limited scope drastically reduces the estimated throughput of a completed manual system to 5.12 req/sec. This suggests that adding the missing functional and non-functional requirements would likely significantly decrease the manual system's throughput. Both systems achieved a 0% error rate during the test, indicating basic operational correctness within their respective implemented scopes. The key takeaway is a trade-off: the incomplete manual system was faster but less functional, while the complete generated system handled comparable load with higher latency due to its comprehensive nature.

4) RESOURCE USAGE

Resource consumption was monitored during the JMeter load test. Under load, the FSMicroGenerator-based implementation consumed more resources in absolute terms (136% CPU, 1292 MB RAM) compared to the manual implementation (19% CPU, 642 MB RAM). This is expected, as FSMicroGenerator runs a more extensive system with 16 containers, compared to the 9 containers used in the manual setup, and implements a full feature set. When normalizing the manual implementation's resource usage, the estimated values rise significantly (96% CPU, 3189 MB RAM), suggesting that a fully implemented manual system would likely demand more resources, especially in terms of RAM. This difference aligns with the architectural distinction: FSMicroGenerator's higher resource usage supports a complete system distributed across more containers, indicating a more granular architectural separation. While the normalized figures for the manual implementation are speculative, they highlight that the generator's architecture inherently facilitates better independent scaling and optimization of components, leading to more efficient resource use per unit of functionality.

5) SUMMARY OF QUANTITATIVE FINDINGS

The quantitative results, when interpreted alongside the normalization estimates, suggest that FSMicroGenerator enabled the junior development team to achieve a complete system within the time limit, requiring significantly less estimated effort and manual intervention compared to the projected requirements for the manual approach involving a more experienced team. FSMicroGenerator produced a larger, more feature-rich codebase. Performance analysis highlights a trade-off between the raw speed of the incomplete manual system and the higher latency but greater functional depth of the complete generated system. Resource usage patterns follow a similar trend, with the auto-generated system consuming more in absolute terms but potentially less per unit of functionality compared to the normalized estimates for the manual approach. These findings indicate that FSMicroGenerator can help less experienced teams deliver complete, production-ready systems within shorter timeframes, with reduced effort, lower risk of structural errors, and more consistent resource usage under realistic conditions.

B. QUALITATIVE ANALYSIS

A qualitative evaluation assessed architectural and structural quality based on the rubric in Appendix C (Table 7), focusing on modularity, reusability, clarity/consistency, and maintainability. Two independent reviewers assigned scores (1-5 scale), resolving significant discrepancies through discussion. The average scores are presented in Table 3.

TABLE 3. Qualitative evaluation (average Likert scores).

Criterion	Manual	FSMicroGenerator
Architectural modularity	3	5
Code reusability	2	5
Code clarity and structure	2	5
Maintainability	2	5

1) ARCHITECTURAL MODULARITY

The evaluation revealed significant differences in architectural modularity between the two implementations. The manual implementation received a moderate score (3/5). Reviewers noted an intention towards modularity (separate frontend, gateway, backend components) but found the implementation inconsistent. Specifically, some services (e.g., inventory) lacked proper internal structure, containing only basic files, whereas others showed more developed organization. This inconsistency raised concerns regarding the separation of responsibilities, as data definitions, business logic, and the communication layer were potentially mixed within single files.

In contrast, the FSMicroGenerator-based implementation achieved an excellent score (5/5). Reviewers highlighted its fully modular system featuring clear separation between the frontend, gateway, and numerous fine-grained backend microservices. Each microservice maintained a consistent internal structure aligned with distinct business capabilities, representing well-defined bounded contexts with loose coupling. The inclusion of separate monitoring and utility components further strengthened the modularity assessment.

2) CODE REUSABILITY

Code reusability assessments also showed marked differences. The manual implementation scored poorly (2/5), with reviewers identifying limited reuse in both frontend and backend components. Although a shared directory existed in the frontend, it primarily contained basic services and models, lacking evidence of complex, reusable UI components (e.g., generic tables). The backend showed minimal use of shared internal patterns or generic functions. Logic was frequently duplicated across components, suggesting high redundancy and limited reuse.

The FSMicroGenerator-based implementation demonstrated strong reusability. Its frontend featured extensive reuse via a comprehensive shared directory containing reusable UI components, directives, models, pipes, resolvers, and services abstracting common logic, a point highlighted by both reviewers. While backend microservices maintained their independence (a characteristic of microservice architecture), the uniform structure across all services, leveraging reusable helper functions and middleware within each service, promoted consistent patterns and reduced redundancy, justifying the assigned score (5/5).

3) CODE CLARITY, STRUCTURAL CONSISTENCY, AND STANDARD CONFORMANCE

The evaluation of code clarity and structural consistency revealed a substantial gap between the implementations. The manual implementation received a poor score (2/5). Reviewers noted significant inconsistencies in both backend and frontend components. Backend services exhibited varying structures, suggesting disparate approaches and development styles. Frontend inconsistencies included an unclear distinction between core and shared folders and mixed component styles (some defining TypeScript, HTML, and CSS inline, e.g., 'inventory.component.ts', while others used separate files). Furthermore, unconventional naming conventions were observed for files, folders, and variables (e.g., microservice folder names like CustomerManagementService, productManagement, supplierService, and transactions).

Conversely, the FSMicroGenerator-based implementation achieved a high score in this category (5/5). Reviewers noted its exceptional consistency and adherence to standards throughout the codebase. All backend microservices followed identical internal directory structures with clear, consistent naming conventions for folders, files, functions, and variables. The frontend maintained similarly high structural consistency. This uniformity, as noted by reviewers, resulted in a coherent codebase that was easy to navigate and understand.

4) MAINTAINABILITY

Maintainability assessments reflected the cumulative impact of other criteria. The manual implementation scored poorly (2/5). Reviewers concluded that its moderate architectural modularity, poor code reusability, and inconsistent structure would impede long-term maintenance. Changes in one area could unexpectedly affect underdeveloped or coupled services, and bug fixes or enhancements would likely necessitate modifications in multiple locations. The lack of backend tests further increased modification risks. Moreover, the absence of comprehensive documentation and an unclear code structure would hinder new developers' ability to quickly understand and contribute, significantly increasing onboarding time.

In contrast, the FSMicroGenerator-based implementation demonstrated excellent maintainability (5/5). Its strong modularity, good reusability, and code clarity provided a foundation for straightforward adaptation and low-risk changes. Reviewers specifically noted that the presence of backend test suites within each microservice significantly reduced modification risks and ensured regressions could be detected early. The clear separation of responsibilities enables independent updates and scaling, while the consistent structure and naming conventions reduce the effort required to understand and modify the code. Additionally, the comprehensive documentation generated for the system further eases the developer onboarding process, facilitating rapid contribution from new team members.

5) DISCUSSION OF QUALITATIVE FINDINGS

The qualitative assessment consistently favored the FSMicroGenerator implementation across all criteria, supported by specific reviewer observations. Notably, both reviewers independently assigned identical scores for all criteria. Although a procedure existed to resolve discrepancies via discussion, no significant disagreements occurred, suggesting a strong consensus on the observed qualitative differences.

FSMicroGenerator appears effective in enforcing architectural best practices, consistency, and generating supporting elements (e.g., tests, documentation) that positively impact maintainability. The results indicate that FSMicroGenerator provides a robust structural foundation—illustrated by uniformity in design, structural consistency across modules, and reusability—which could be challenging to achieve reliably through manual development efforts.

C. REAL WORLD ADOPTION

Beyond the comparative evaluation, FSMicroGenerator has been successfully deployed in production environments. It was used by Smart Automation Technologies, a Moroccan startup, in the development of several real projects, including Smart DLSP (Smart Digital Logistic Services Providers)—a national initiative under the Al Khawarizmi program, aimed at modernizing freight and logistics operations through intelligent and AI-powered solutions. In this context, FSMicroGenerator was used to automatically generate the full-stack architecture of a digital freight exchange platform, including modular backend microservices and Angular-based frontend components. The development was structured in two stages: first, delivering a functional and scalable base system for real-time logistics flow management; then, enabling the future integration of AI agents to automate decision-making processes. The tool played a key role in supporting this phased approach, by ensuring architectural consistency, reducing development time, and facilitating the seamless extension of the platform with intelligent capabilities.

VII. DISCUSSION

The evaluation indicates that FSMicroGenerator has a measurable impact on optimizing software development, especially within complete microservices architectures. This section outlines the key benefits observed, assesses alternative approaches, and discusses the main limitations identified during its use.

A. IMPACT OF FSMICROGENERATOR ON SOFTWARE DEVELOPMENT

FSMicroGenerator functions as a code generation tool integrated into an ecosystem designed to reduce technical barriers and facilitate collaboration between development and operations teams. By automating not only source code generation but also the preparation of deployment environments, infrastructure configuration, and service monitoring, it allows developers to concentrate more on business aspects rather than time-consuming technical tasks. This approach can lead to increased productivity while providing a structured framework for the development of scalable solutions adapted to different business contexts.

Another key benefit observed relates to the standardization and quality of the code generated. Through the adoption of OMG standards, RESTful best practices, and DevSecOps principles, FSMicroGenerator generates homogeneous, well-architected solutions aligned with modern standards. The use of design patterns, generic functions, and reusable components contributes to improved application maintainability and scalability.

FSMicroGenerator appears to improve accessibility by simplifying microservices adoption, especially for teams less experienced with distributed systems or infrastructure automation. Its use of a JavaScript-based stack reduces context-switching between frontend and backend technologies, allowing developers to focus on business logic rather than infrastructure concerns. These characteristics make FSMicroGenerator particularly suitable in settings affected by talent shortages, helping teams produce well-structured architectures without significantly impacting timelines or costs.

B. ASSESSMENT OF ALTERNATIVE APPROACHES

Unlike traditional development, which relies on a manual, often tedious, and error-prone process, FSMicroGenerator adopts an automated approach for rapid solution generation. This method promotes code consistency and accelerates the transition from design to deployment.

Compared with low-code/no-code solutions, FSMicroGenerator offers greater flexibility by enabling advanced customization. Whereas LCNC platforms often impose restrictions on the architecture and languages used, the generated code is fully exploitable and modifiable, enabling development teams to integrate specific functionalities and technical optimizations tailored to business needs.

Unlike tools based on generative AI, which mainly produce portions of code without providing a complete solution, FSMicroGenerator is designed to generate entire architectures tailored to the specific needs of a project. What's more, while GenAI tools are effective as assistance tools, their limitations in terms of security and compliance require particular attention.

Finally, FSMicroGenerator differs from other code generators due to its modular approach, microservices orientation, and native integration of best practices. It is not limited to CRUD generation, but covers aspects across the entire application development cycle. It also features the capability to build a catalog of packages and modules to feed projects, promoting extensibility, reusability and potentially faster delivery of IT solutions.

Table 4 provides a comparative analysis of the different code generation tools discussed in Section II and FSMicroGenerator, emphasizing their distinctive features and capabilities.

C. IDENTIFIED LIMITATIONS AND CHALLENGES

Despite the observed benefits, FSMicroGenerator has certain limitations that need to be taken into account in order to refine its use.

TABLE 4. Comparative analysis of FSMicroGenerator and existing code generation tools.

Criteria	Uyanik & Sahin [1]	Ullah & Inayat [6]	Livraghi [17]	Paolone et al. [18]	Ražinskas & Čeponienė [19]	Prasanthan et al. [2]	Tesoriero et al. [20]	Our solution
Scope	Full-stack	Backend + integrated UI	Full-stack	Backend + integrated UI	Backend + integrated UI	Backend	Full-stack	Full-stack
Architecture	Monolithic	Monolithic	Monolithic	Monolithic	Monolithic	Microservices	Monolithic	Microservices
Target languages	C# (ASP.NET MVC), JavaScript (Handlebars)	Python	Java (Spring), JavaScript (AngularJS)	Java (Hibernate, Vaadin)	PHP (Laravel)	JavaScript (Node.js)	PHP (Propel ORM), JavaScript (Polymer, HTML)	JavaScript (MEAN stack)
Scalability	Medium	Low	Medium	Low	Low	High	Medium	High
Deployment (containerization)	No	No	No	No	No	Yes	No	Yes
Interoperability (integration with external systems)	Moderate	Difficult	Moderate	Difficult	Difficult	Easy	Moderate	Easy
Support multilingual	No	No	No	No	No	-	No	Yes
User experience (documentation, learning curve, ease of use)	Medium	Medium	Good	Medium	Medium	Medium	Medium	Very good
Monitoring (logging, performance tracking, system health)	No	No	No	No	No	No	No	Yes
Security (authentication, role management, data validation)	Low	Low	High	Low	Low	Medium	Low	Very high
Business operations	CRUD	CRUD	CRUD	CRUD	CRUD	CRUD	CRUD	CRUD, translation, clone

One of the main aspects is the creation stage in the package lifecycle. Currently, this stage is based on a semi-manual approach, requiring the user to manually define business entities via the forms provided by the tool, according to the UML class diagram. Although this approach offers flexibility for defining business models, it introduces human intervention that can generate errors or slow down the initial configuration, especially for large-scale projects.

Another challenge identified is the management of advanced customization. Although the generator offers a standardized, modular structure, some specific use cases may require manual adjustments after initial generation, notably the integration of complex business functionality and advanced user interface customization. This necessitates human intervention to adapt the generated code to the specific needs of the project during the adaptation phase.

Finally, although the tool ensures effective standardization, it might induce a certain dependency on the generated models, which could make the integration of new technologies or frameworks not supported by the generator a little trickier. However, this constraint is offset by the modularity of the code produced, which remains modifiable and extensible by developers.

VIII. CONCLUSION AND PERSPECTIVES

The results suggest that FSMicroGenerator is suitable for automating the development of complex microservice architectures. It can reduce implementation times, improve interoperability, and support system maintainability.

Its adoption in large-scale projects demonstrates flexibility and adaptability across various technical contexts, including those involving AI integration. By automating the generation of multilingual full-stack solutions, from development to deployment, with integrated monitoring and supervision, and by applying widely accepted development practices, it helps lower technical barriers and reduce technical debt. This enables teams to focus on domain-specific functionality while adopting modern, advanced, and proven technologies. It also promotes compliance with software engineering standards and mitigates risks related to architectural inconsistency. Beyond the observed time savings, FSMicroGenerator contributes to improving scalability, resilience, and performance in enterprise software systems.

Although FSMicroGenerator has been applied in diverse real-world cases, ongoing research aims to extend its capabilities and evaluate its performance in additional domains.

- **Advanced orchestration:** FSMicroGenerator currently uses Docker Compose for local microservices management. Integrating Kubernetes would allow for easy scaling of container replicas across multiple physical hosts, improving resource management, load balancing, and automatic scalability. Kubernetes enhances resilience and cost optimization, particularly in complex production environments.
- **AI-driven automation:** The integration of AI would enable automatic transformation of class diagrams (in image, XML, JSON, or Mermaid format) or natural language business specifications into usable inputs for FSMicroGenerator. This would improve the semi-automatic process during the package creation stage, reduce human error, and ensure real-time compliance.
- **Micro-frontend adoption:** The application of the micro-frontend principle would further decompose the user interface, facilitating complexity management and improving application responsiveness and scalability.
- **Support for new technology stacks:** By extending support to additional JavaScript stacks such as MERN and MEVN, FSMicroGenerator would offer greater flexibility, enabling teams to choose the technology best suited to their projects.
- **Mobile adaptability:** Integrating architectures adapted to mobile applications would enable the creation of scalable, high-performance solutions, meeting the growing demand for mobile applications without sacrificing backend quality.
- **Open source:** Making FSMicroGenerator open source would foster collaborative innovation, enabling a worldwide community to contribute to its development, enrich its functionalities, and reinforce its durability.

These potential enhancements could significantly impact the optimization of development processes and improve efficiency, enabling FSMicroGenerator to become an even more complete and adaptable solution, capable of meeting the challenges of an ever-changing technological environment.

APPENDIX A
FUNCTIONAL COVERAGE CHECKLIST

Table 5 provides a binary evaluation matrix for each functional feature expected in the experimental case study. This checklist was used to calculate the **functional coverage ratio** (R_f) by verifying whether each item was implemented or missing in a given solution.

Note: Functionalities marked with an asterisk (*) are automatically generated by FSMicroGenerator, including basic CRUD operations, the setup of entity relationships (e.g., foreign keys, form bindings, API endpoints, service scaffolding), and filtering logic. Other features, especially those involving business rules, dynamic constraints, or integration workflows, were intentionally included to assess whether the tool facilitates or accelerates their implementation by developers. The distinction is crucial to evaluate not only what FSMicroGenerator produces directly, but also how it supports the development of advanced logic through clean structure, reusable patterns, and architectural consistency.

APPENDIX B
NON-FUNCTIONAL COVERAGE CHECKLIST

Table 6 presents the set of non-functional requirements (NFRs) defined for the case study. These NFRs encompass essential cross-cutting concerns, including security, data integrity, observability, deployment automation, and user experience.

Rather than using a binary presence check at the global system level, we adopted a two-level evaluation strategy to compute the non-functional coverage ratio (R_{nfr}):

- **Module-level NFRs**: For requirements such as security, error handling, caching, logging, internationalization, and metrics exposure (monitoring), coverage was assessed *per functional module*. In these cases, the coverage value corresponds to the ratio of functional modules that correctly implemented the requirement.
- **System-wide NFRs**: For system-wide requirements such as CI/CD and Docker-based orchestration, a global binary evaluation was conducted. However, each module was required to meet its corresponding integration responsibilities, such as contributing unit tests to the CI pipeline, and defining its deployment parameters within the orchestration configuration.

This distinction ensures precise and equitable computation of R_{nfr}, reflecting both the distributed and centralized aspects of non-functional integration. All NFRs listed in Table 6 were automatically handled by FSMicroGenerator, either globally or per module, demonstrating its ability to enforce non-functional consistency while reducing developer effort on cross-cutting concerns.

APPENDIX C
RUBRIC FOR QUALITATIVE EVALUATION

Table 7 summarizes the rubric used to evaluate qualitative metrics. It provides a standardized evaluation framework to guide reviewers in assessing key architectural aspects of both generated and manually developed applications. The rubric focuses on four dimensions: modularity, reusability, clarity, and maintainability. Each is rated on a consistent 5-point scale with predefined criteria. While the rubric helps reduce subjectivity and variation across reviewers, it still allows for expert judgment where interpretation is needed, ensuring fair and comparable assessments grounded in a shared evaluation baseline.

TABLE 5. Checklist of functional requirements per module (for R_f evaluation).

Functional module	Feature description	Manual impl.	FSMicroGenerator
Product management	CRUD operations for Product entity*	1	1
	Automatic product status update	0	1
	Visualize inventory movements related to a specific product*	0	1
	Search, filter and sort functionalities*	1	1
Category management	CRUD operations for Category entity*	1	1
	Support hierarchical structure (1..*)*	0	1
	Display products of a specific category*	0	1
Inventory movement	Disallow manual create/update of movements (must be triggered automatically)	1	1
	Automatically trigger stock adjustment based on events	1	1
	Filter inventory movements by type and date*	1	1
Transaction	CRUD operations for Transaction entity*	1	1
	Calculate total amount	1	1
	Handle currency exchange in total calculation	0	1
	Trigger inventory movement based on transaction type/status	1	1
	Manage customer/supplier relationship per transaction type	1	1
	Search, filter and sort functionalities*	1	1
Transaction line	CRUD operations for TransactionLine entity*	1	1
	Calculate line totals	1	1
	Enforce stock constraints in sales(quantity $\leq$ available stock)	1	1
	Restrict product selection based on transaction type and supplier	1	1
Supplier management	CRUD operations for Supplier entity*	1	1
	List supplied products*	0	1
	List supplier-related transactions*	0	1
Customer management	CRUD operations for Customer entity*	1	1
	List customer-related transactions*	0	1
External integration	Integrate taxonomy API for classification references*	0	1
	Integrate currency API to support multi-currency pricing and exchange rate conversion*	0	1
	Integrate account package for authentication and role based access*	0	1
	Total implemented features	**17**	**28**
	Functional coverage ratio (R_f)	**17/28 = 0.61**	**28/28 = 1.00**

* Feature automatically generated by FSMicroGenerator.

TABLE 6. Checklist of non-functional requirements with evaluation scope and coverage.

Scope	Category	Requirement description	Manual impl.	FSMicroGenerator
Module-level	Internationalization*	Dynamic label translation and user language preference management	0/5	5/5
	User feedback on operation*	Display of toast/notification messages for user actions (success/failure)	0/5	5/5
	Input validation (client-side)*	Real-time validation of user inputs and forms before submission	5/5	5/5
	Security (access control)*	Authentication, authorization, and rate limiting	0/5	5/5
	Input Validation (Server-side)*	Validation of payloads and parameters at backend level	1/5	5/5
	Caching*	Redis caching for frequently accessed endpoints	0/5	5/5
	Error handling*	Standard HTTP status codes and structured error messages	5/5	5/5
	Logging*	Centralized error and request logging, structured and persisted logs	5/5	5/5
	Monitoring*	Prometheus metrics and Grafana dashboards; each module exposes a /metrics endpoint	0/5	5/5
	API documentation*	Swagger/OpenAPI documentation generated for all endpoints per module	0/5	5/5
System-level	Deployment & Orchestration*	Docker containerization and orchestration via Docker Compose	1/1	1/1
	CI/CD*	GitLab pipeline for test, build, and deployment stages	0/1	1/1
		Total NFRs covered	**17**	**52**
		Non-functional coverage ratio (R_{nfr})	**17/52 = 0.33**	**52/52 = 1.00**

* Feature automatically generated by FSMicroGenerator.

TABLE 7. Structured rubric for qualitative metrics (5-point Likert scale).

Metric	Very poor (1)	Poor (2)	Moderate (3)	Good (4)	Excellent (5)
Architectural modularity	No separation of concerns; modules are tightly coupled and monolithic	Minimal separation; only superficial layering; responsibilities are mixed	Partial modularity; some services isolated but overlap remains	Clear decomposition into logically separated services; minor coupling remains	Fully modular system; well-aligned bounded contexts; cohesive and loosely coupled
Code reusability	Business logic is duplicated; no shared components	Few reusable parts; high redundancy	Some reuse across modules; basic abstraction present	Good use of shared libraries or generic functions; minimal repetition	Extensive reuse via abstracted, generic components; no duplication
Code clarity, structural consistency, and standard conformance	Unreadable code; no naming or structural conventions; folders disorganized	Inconsistent naming; some conventions applied but poorly followed	Mostly consistent structure and naming; some unclear areas remain	Clean and consistent structure; good adherence to naming and design conventions	Fully consistent and well-structured codebase; strong conformance to patterns (e.g., MVC, RESTful)
Maintainability	Code is difficult to understand or update; tightly coupled; prone to regressions	Fragile and risky to modify; poor separation of concerns	Reasonable modularity; maintainability possible with moderate effort	Good modularity and testability; low-risk changes are feasible	Highly maintainable architecture; clean separation of concerns and easy adaptation

CODE AVAILABILITY

The source code used in the comparative case study—covering both FSMicroGenerator-generated and manually developed inventory management systems—is publicly available on GitHub at https://github.com/samiraKHALFAOUI/fsmicrogenerator-evaluation

REFERENCES

[1] B. Uyanik and V. H. Şahin, "A template-based code generator for Web applications," *TURKISH J. Electr. Eng. Comput. Sci.*, vol. 28, no. 3, pp. 1747–1762, May 2020, doi: 10.3906/elk-1910-44.

[2] A. Prasanthan, K. S. Anand, B. P. Nair, K. Gautham Santhosh, and J. Swaminathan, "Low code backend as a service platform," in *Proc. World Conf. Commun. Comput. (WCONF)*, Jul. 2023, pp. 1–8.

[3] K. Shinde and Y. Sun, "Template-based code generation framework for data-driven software development," in *Proc. 4th Int. Conf Appl. Comput. Inf. Technol./3rd Int. Conf Comput. Sci./Intell. Appl. Inform./1st Int. Conf Big Data, Cloud Comput., Data Sci. Eng. (ACIT-CSII-BCD)*, Dec. 2016, pp. 55–60.

[4] S. Rädler, L. Berardinelli, K. Winter, A. Rahimi, and S. Rinderle-Ma, "Bridging MDE and AI: A systematic review of domain-specific languages and model-driven practices in AI software systems engineering," *Softw. Syst. Model.*, vol. 24, no. 2, pp. 445–469, Apr. 2025, doi: 10.1007/s10270-024-01211-y.

[5] E. Syriani, L. Luhunu, and H. Sahraoui, "Systematic mapping study of template-based code generation," *Comput. Lang., Syst. Struct.*, vol. 52, pp. 43–62, Jun. 2018, doi: 10.1016/j.cl.2017.11.003.

[6] I. Ullah and I. Inayat, "Template-based automatic code generation for Web application and APIs using class diagram," in *Proc. Int. Conf. Frontiers Inf. Technol. (FIT)*, Dec. 2022, pp. 332–337.

[7] L. Luhunu and E. Syriani, "Comparison of the expressiveness and performance of template-based code generation tools," in *Proc. 10th ACM SIGPLAN Int. Conf. Softw. Lang. Eng.*, Oct. 2017, pp. 206–216.

[8] S. Muhammad, V. Prybutok, and V. Sinha, "Unlocking citizen developer potential: A systematic review and model for digital transformation," *Encyclopedia*, vol. 5, no. 1, p. 36, Mar. 2025, doi: 10.3390/encyclopedia5010036.

[9] M.-F. Wong, S. Guo, C.-N. Hang, S.-W. Ho, and C.-W. Tan, "Natural language generation and understanding of big code for AI-assisted programming: A review," *Entropy*, vol. 25, no. 6, p. 888, Jun. 2023, doi: 10.3390/e25060888.

[10] S. Kotsiantis, V. Verykios, and M. Tzagarakis, "AI-assisted programming tasks using code embeddings and transformers," *Electronics*, vol. 13, no. 4, p. 767, Feb. 2024, doi: 10.3390/electronics13040767.

[11] A. Odeh, N. Odeh, and A. S. Mohammed, "A comparative review of AI techniques for automated code generation in software development: Advancements, challenges, and future directions," *TEM J.*, VOL. 13, pp. 726–739, Feb. 2024, doi: 10.18421/tem131-76.

[12] E. Martinez and L. Pfister, "Benefits and limitations of using low-code development to support digitalization in the construction industry," *Autom. Construct.*, vol. 152, Aug. 2023, Art. no. 104909, doi: 10.1016/j.autcon.2023.104909.

[13] S. Haag and A. Eckhardt, "Dealing effectively with shadow IT by managing both cybersecurity and user needs," *MIS Quart. Executive*, vol. 23, no. 4, pp. 399–412, 2024, doi: 10.17705/2msqe.00104.

[14] T. H. Davenport, "MISQE insight: On the inevitability of citizen development," *MIS Quart. Executive*, vol. 22, no. 4, pp. 1–10, 2023.

[15] C. Negri-Ribalta, R. Geraud-Stewart, A. Sergeeva, and G. Lenzini, "A systematic literature review on the impact of AI models on the security of code generation," *Frontiers Big Data*, vol. 7, May 2024, Art. no. 1386720, doi: 10.3389/fdata.2024.1386720.

[16] M. Hamza, D. Siemon, M. A. Akbar, and T. Rahman, "Human-AI collaboration in software engineering: Lessons learned from a hands-on workshop," in *Proc. 7th ACM/IEEE Int. Workshop Softw.-Intensive Bus.*, Apr. 2024, pp. 7–14.

[17] M. Livraghi and M. Brambilla, "Automatic generation of web CRUD applications," M.S. thesis, Dept. Master Sci. Comput. Eng., Politecnico di Milano, Milan, Italy, 2016. Accessed: Jun. 6, 2025. [Online]. Available: https://www.politesi.polimi.it/bitstream/10589/125742/1/2016_09_Livraghi.pdf

[18] G. Paolone, M. Marinelli, R. Paesani, and P. Di Felice, "Automatic code generation of MVC Web applications," *Computers*, vol. 9, no. 3, p. 56, Jul. 2020, doi: 10.3390/computers9030056.

[19] M. Ražinskas and L. Čeponienė, "MDA approach for laravel framework code generation from UML diagrams," in *Proc. CEUR Workshop*, vol. 2698, Jan. 2020, pp. 106–113.

[20] R. Tesoriero, A. Rueda, J. A. Gallud, M. D. Lozano, and A. Fernando, "Transformation architecture for multi-layered WebApp source code generation," *IEEE Access*, vol. 10, pp. 5223–5237, 2022, doi: 10.1109/ACCESS.2022.3141702.

[21] B. Hippchen, P. Giessler, R. H. Steinegger, M. Schneider, and S. Abeck, "Designing microservice-based applications by using a domain-driven design approach," *Int. J. Adv. Softw.*, vol. 10, pp. 432–445, Dec. 2017.

[22] K. Gos and W. Zabierowski, "The comparison of microservice and monolithic architecture," in *Proc. IEEE 16th Int. Conf. Perspective Technol. Methods MEMS Design (MEMSTECH)*, Apr. 2020, pp. 150–153, doi: 10.1109/MEMSTECH49584.2020.9109514.

[23] P. S. Samant, "Microservices in the cloud: Enabling scalability, flexibility, and rapid deployment," *J. Adv. Res. Eng. Technol.*, vol. 3, no. 1, pp. 1–10, 2024.

[24] V. Velepucha and P. Flores, "A survey on microservices architecture: Principles, patterns and migration challenges," *IEEE Access*, vol. 11, pp. 88339–88358, 2023, doi: 10.1109/ACCESS.2023.3305687.
[25] I. Karabey Aksakalli, T. Çelik, A. B. Can, and B. Tekinerdoğan, "Deployment and communication patterns in microservice architectures: A systematic literature review," *J. Syst. Softw.*, vol. 180, Oct. 2021, Art. no. 111014, doi: 10.1016/j.jss.2021.111014.
[26] I. K. Aksakalli, T. Celik, A. B. Can, and B. Tekinerdogan, "Systematic approach for generation of feasible deployment alternatives for microservices," *IEEE Access*, vol. 9, pp. 29505–29529, 2021, doi: 10.1109/ACCESS.2021.3057582.
[27] P. Porter, S. Yang, and X. Xi, "The design and implementation of a RESTful IoT service using the MERN stack," in *Proc. IEEE 16th Int. Conf. Mobile Ad Hoc Sensor Syst. Workshops (MASSW)*, Nov. 2019, pp. 140–145.
[28] Stack Overflow. *Developer Survey*. Accessed: Jun. 6, 2025. [Online]. Available: https://survey.stackoverflow.co/2023/
[29] B. S. Kim, S. H. Lee, Y. R. Lee, Y. H. Park, and J. Jeong, "Design and implementation of cloud Docker application architecture based on machine learning in container management for smart manufacturing," *Appl. Sci.*, vol. 12, no. 13, p. 6737, Jul. 2022, doi: 10.3390/app12136737.
[30] B. Dunka, E. Emmanuel, and D. O. Oyerinde, "Simplifying web application development using MEAN stack technologies," *Int. J. Latest Res. Eng. Technol.*, vol. 2018, pp. 520–531, Jan. 2018.
[31] N. Nikolakis, A. Marguglio, G. Veneziano, P. Greco, S. Panicucci, T. Cerquitelli, E. Macii, S. Andolina, and K. Alexopoulos, "A microservice architecture for predictive analytics in manufacturing," *Proc. Manuf.*, vol. 51, pp. 1091–1097, Jan. 2020, doi: 10.1016/j.promfg.2020.10.153.
[32] Y. Jani, "Unified monitoring for microservices: Implementing prometheus and grafana for scalable solutions," *J. Artif. Intell., Mach. Learn. Data Sci.*, vol. 2, no. 1, pp. 848–852, Mar. 2024, doi: 10.51219/jaimld/yash-jani/206.
[33] A. Martnez and C. Rivera, "Enhancing reliability through effective system monitoring," *Eigenpub Rev. Sci. Technol.*, vol. 8, no. 7, pp. 1–10, 2024.
[34] L. Giamattei, A. Guerriero, R. Pietrantuono, S. Russo, I. Malavolta, T. Islam, M. Dînga, A. Koziolek, S. Singh, M. Armbruster, J. M. Gutierrez-Martinez, S. Caro-Alvaro, D. Rodriguez, S. Weber, J. Henss, E. F. Vogelin, and F. S. Panojo, "Monitoring tools for DevOps and microservices: A systematic grey literature review," *J. Syst. Softw.*, vol. 208, Feb. 2024, Art. no. 111906, doi: 10.1016/j.jss.2023.111906.

SAMIRA KHALFAOUI received the State Engineering Diploma degree in information systems from the National School of Applied Sciences of Tangier, Morocco, in 2021. She is currently pursuing the Ph.D. degree in computer science, focusing on software engineering. Her research explores the integration of artificial intelligence into software engineering to automate and accelerate software lifecycle management. Drawing on her academic background and professional experience, her work aims to simplify complex development processes, minimize technical barriers, reduce manual intervention, and enhance scalability in modern IT systems. By leveraging AI, combining model-driven approaches, DevSecOps practices, and intelligent code generation, she seeks to bridge the gap between innovative architectures and practical industry adoption, empowering developers to build agile, and maintainable distributed systems.

HAFIDA KHALFAOUI received the B.Sc. degree in electronic and telecommunication engineering and the M.Sc. degree in telecommunication systems and computer networks from Sultan Moulay Slimane University, Beni Mellal, Morocco, in 2017 and 2019, respectively, and the Ph.D. degree in computer networks and security from the Polydisciplinary Faculty of Beni Mellal, Morocco, in 2024. Her research interests include computer science, networking, and security.

ABDELLAH AZMANI received the Ph.D. degree in industrial computing in dynamic system modeling and artificial intelligence from the University of Science and Technology of Lille, in 1991. He was a Professor with the Ecole Centrale of Lille, France, and the Institute of Computer and Industrial Engineering, Lens, France. He is currently a Professor with the Faculty of Science and Technology of Tangier, Morocco. He is the Director of the Intelligent Automation and BioMedGenomics Laboratory and created the Intelligent Automation Team, which he coordinates. He has contributed to many theses and scientific research projects, and he elaborates and produces many IT and decision support solutions for public administration, business management, marketing, and logistics.