RI: A DSL for Software Development Project Continuous Configuration

Cássio Ritse Machado Dos Santos Silva
Instituto Politécnico de Bragança, Portugal

Maria João Varanda Pereira
Research Centre in Digitalization and Intelligent Robotics (CeDRI),
Laboratório Associado para a Sustentabilidade e Tecnologia em Regiões de Montanha (SusTEC),
Instituto Politécnico de Bragança, Portugal

Paulo Alexandre Vara Alves
Research Centre in Digitalization and Intelligent Robotics (CeDRI),
Laboratório Associado para a Sustentabilidade e Tecnologia em Regiões de Montanha (SusTEC),
Instituto Politécnico de Bragança, Portugal

Gustavo Jansen de Souza Santos
Universidade Tecnológica Federal do Paraná (UTFPR), Dois Vizinhos, Brazil

Abstract

This paper addresses the challenges related to standardization and automation in the development and maintenance of microservice-based architectures, particularly in heterogeneous environments involving multiple development teams. In such contexts, the absence of unified standards for service definition and implementation, combined with technological diversity and team fragmentation, can lead to architectural inconsistencies, increased integration effort, difficulties in system evolution, and higher maintenance costs over time. Additionally, a significant portion of configuration, orchestration, and artifact generation tasks is performed manually, increasing the risk of errors and rework. As a solution, this paper proposes a Domain-Specific Language (DSL), named RI language, designed for the declarative description of microservices, along with a supporting tool, TSE (Toolbox Service Executor), implemented in Python (PLY) and based on the RI grammar. The tool is responsible for interpreting specifications written in RI and automating the generation of software artifacts through extensible plugins. The proposed approach enables the representation of structural and operational aspects of services in a technology-agnostic manner, promoting consistency, reuse, and reduction of manual effort.

2012 ACM Subject Classification Software and its engineering → Domain specific languages

Keywords and phrases DSL, microservices, software configuration, code generator

Digital Object Identifier 10.4230/OASIcs.SLATE.2026.21

Funding This work was supported by FCT – Fundação para a Ciência e Tecnologia, I.P. by projects: CeDRI, UID/05757/2025 (DOI: `https://doi.org/10.54499/UID/05757/2025`) and UID/-PRR/05757/2025 (DOI: `https://doi.org/10.54499/UID/PRR/05757/2025`); SusTEC, LA/P/0007/2020 (DOI: `https://doi.org/10.54499/LA/P/0007/2020`).

Acknowledgements The authors would also like to thank the supervising professors and co-authors for their guidance and collaboration, as well as the Universidade Tecnológica Federal do Paraná (UTFPR), Brazil, and the Instituto Politécnico de Bragança (IPB), Portugal, for their institutional support.

1 Introduction

Microservice architecture has been widely adopted in the development of modern distributed systems, primarily due to its ability to promote scalability, modularity, and independence among components. However, as the number of services increases, so does the complexity associated with standardization, integration, and maintenance of these systems [9].

One of the main challenges in this context is ensuring consistency in how different microservices are structured and exposed, especially in environments involving multiple teams and technologies. Even when different teams use the same programming language and similar frameworks, it is common for them to adopt distinct implementation patterns for aspects such as code organization, route definitions, service configuration, and communication strategies [6]. These variations, although often subtle, can lead to significant inconsistencies, making system maintenance, evolution, and service integration more difficult. For example, different teams may adopt REST (Representational State Transfer) conventions with explicit versioning in the URL (e.g., `/v1/users`), while others may omit versioning or use different naming and routing structures. Additionally, differences in response formats may require specific adaptations on the consumer side, increasing coupling and integration complexity.

Although each microservice maintains autonomy over its business logic implementation, aspects such as communication, configuration, structural organization, and implementation patterns often require standardization [6]. The absence of clear guidelines for these elements can result in rework and increased maintenance effort.

Given this scenario, it becomes relevant to define mechanisms that allow the standardized and technology-agnostic description of structural and operational characteristics of software projects. In this context, solutions based on Domain-Specific Languages (DSLs) and language processing techniques emerge as a promising approach, as they enable the declarative modeling of domain-specific elements [1].

In this work, we propose the RI DSL for the description of software projects, along with the development of a toolbox, TSE, which enables the interpretation of these specifications and the association of semantic actions to automate the generation and configuration of software artifacts. The goal is to increase automation in the integration and configuration of software projects across different teams and technological contexts, in a consistent and standardized manner.

2 Background

A microservice can be defined as an autonomous software component, independently deployed and maintained, whose purpose is to execute complete business logic or perform specific tasks. These services are organized within a microservice-based architecture, in which each component contributes to a broader objective, collectively forming a complete distributed system [4].

One of the fundamental principles of this architecture is the use of standardized service contracts, through which microservices expose their functionalities, capabilities, and interaction policies. These contracts play a crucial role in service communication, ensuring interoperability and loose coupling among system components [11].

Therefore, the ability of a microservice to participate in more complex compositions should be considered from its design phase. This approach helps prevent future rework and ensures that the service can effectively collaborate with other components, contributing to the development of more robust, scalable, and integrated solutions.

2.1 Challenges in Microservice-Based Architectures

In microservice development, it is common to deal with the inherent complexity of managing multiple services alongside the heterogeneity of technologies and tools [8, 10]. Over time, microservice-based applications may exhibit issues related to quality, interoperability, and lack of cohesion among services implemented using different languages and technologies.

The use of API gateways or messaging systems can simplify communication and coordination between independent microservices; however, it introduces significant challenges in terms of configuration and implementation. Configuring an API (Application Programming Interface) gateway requires the precise definition of routing rules, security policies, and authentication mechanisms, which can become complex and error-prone, especially in dynamic environments where services are frequently added or modified [10]. Similarly, messaging systems require proper configuration of queues, topics, subscriptions, and message handlers, which can be difficult to manage in medium- and large-scale systems.

Ensuring compatibility across different API versions is also a challenge, requiring strict versioning and deprecation strategies [10]. Moreover, effective communication depends on well-defined and up-to-date documentation, enabling developers to correctly follow established standards. The absence of such documentation may lead to configuration errors and difficulties in integrating new services.

The use of containers enables the reuse and deployment of the same microservice across different environments, such as testing, staging, and production. However, each environment typically has specific requirements, including environment variables, database configurations, and network settings. Managing these variations for the same microservice across multiple environments can become a significant operational and deployment challenge.

Microservice architectures naturally promote technological heterogeneity, allowing the integration of services implemented in different programming languages and using diverse communication protocols, which can optimize performance and scalability [10]. However, this diversity also introduces risks that may compromise system cohesion and design consistency [8]. For instance, different teams may adopt conflicting conventions for error handling or communication patterns. Therefore, it is essential to establish common guidelines and practices to ensure consistency and interoperability across services, mitigating the risks associated with technological diversity and preserving the overall integrity of the system.

2.2 Related Work

Given the challenges discussed, there is a clear need for an intermediate representation capable of abstracting structural and operational aspects of microservices in a standardized manner, without compromising the implementation autonomy of each service. Such a representation should act as a structural contract, promoting consistency in how services are defined, organized, and exposed.

Furthermore, the absence of automated mechanisms for applying these standards tends to increase divergence between projects, particularly in heterogeneous environments. In this context, it becomes essential to adopt an approach that not only enables the unified description of such patterns but also supports their automation. Domain-Specific Languages (DSLs) have been identified as a promising alternative, as they allow the declarative and structured modeling of domain-specific elements [7].

Based on a systematic literature review, in which 830 studies were initially identified and 38 were selected for detailed analysis after applying selection criteria, it was observed that few approaches address standardization and automation in an integrated manner within microservice architectures. In particular, there is a gap in solutions that are simultaneously generic, extensible, and technology-agnostic, especially in environments involving multiple languages and frameworks.

Additionally, although existing works explore DSLs and automation, they are mostly focused on specific domains such as the Internet of Things [5], embedded systems, or infrastructure automation [12], and do not directly address the challenges of standardization and integration in microservice-based systems considered in this study.

Among the works that are closer to the proposed approach, two stand out: *Developing Microservice-Based Applications Using the Silvera Domain-Specific Language* [13] and *Blueprint: A Toolchain for Highly-Reconfigurable Microservices* [2].

Silvera [13] is a DSL designed for the declarative modeling of microservice-based architectures, enabling the specification of elements such as services, gateways, and endpoints, from which code can be automatically generated in different programming languages. In contrast, Blueprint [2] proposes a toolchain-based approach for configuring, building, and deploying microservices, leveraging abstractions and an intermediate representation to enable artifact generation and application reconfiguration.

While both approaches demonstrate the use of abstraction and automation to reduce complexity in distributed systems development, they primarily focus on initial code generation and structural configuration. They do not fully address the integrated automation of multiple configuration scenarios throughout the `software lifecycle`, nor do they emphasize generality in heterogeneous environments and the standardization of diverse configuration aspects.

In many practical development environments, standardization processes still depend heavily on manual supervision and on the expectation that all developers consistently follow architectural and organizational conventions previously defined by technical leaders or software architects. This often leads to recurrent verification activities, manual code reorganizations, and inconsistencies between teams and projects, especially in distributed environments. By contrast, the approach proposed in this work seeks to embed these conventions directly into the generation and configuration process, allowing such standards to be automatically applied and propagated across projects. Based on the professional experience of the first author, it is expected that this type of automation can produce short-term gains in consistency and development productivity, although the approach still requires further validation in real industrial environments.

3 RI-Language and TSE

This work proposes two main contributions: (i) the RI language, a Domain-Specific Language (DSL) designed for the declarative description of microservices; and (ii) an associated tool, named TSE, responsible for interpreting RI specifications and automating the generation of software artifacts by associating semantic actions with the grammar productions of the language.

The proposed language enables the structured and technology-independent description of microservice projects, including their properties and specifications. Complementarily, the TSE tool, developed using language processing techniques, leverages the RI lexer and parser to interpret specifications and automate the implementation of modules and configurations according to the defined patterns [3]. This automation is achieved through code generation, file creation, and controlled modification of specific code segments, reducing manual effort and promoting greater consistency across projects.

3.1 RI Language

The RI language was conceived as a Domain-Specific Language (DSL) aimed at the structured description of microservices and their associated artifacts. Its primary goal is to provide a declarative, standardized, and easily interpretable representation. In this way, the language supports not only tool-driven automation but also understanding and manipulation by users with different levels of technical expertise. Additionally, RI serves as a form of living

documentation, facilitating the understanding of microservice structures, their dependencies, and configurations throughout the software lifecycle. Listing 1 presents a simple project declared using the RI language.

Listing 1 Simple project declared in RI language.

```
project "ms-customer" {
  use nestjs
  git "https://github.com/user/ms-customer.git"
  config "default"
  folders clean_architecture
  deployment {
                version "0.0.1"
                port 3000
                url "http://localhost"
        }
  dependencies []
  controllers [
    {
      name "CustomerController"
      path "/customer"
      endpoints[
        {
          type REST
          name "getOneCustomer"
          pattern GET
          path "/:id"
        }
        {
          type REST
          name "createCustomer"
          pattern POST
          path "/"
        }
      ]
    }
  ]
}
```

The central construct of the language is the `project` block, which represents a project. This block encapsulates all relevant information required to describe a service within the tool's context. A project may include general metadata, infrastructure configurations, dependency definitions, and specifications of exposed interfaces. It comprises declarations that describe fundamental characteristics such as the adopted framework (`use`), the remote repository (`git`), predefined configurations (`config`), and the directory organization pattern (`folders`). The `deployment` section groups execution-related information, such as version, port, and base URL.

The `dependencies` section allows explicit declaration of relationships with other projects and plugins. Through this structure, it is possible to define service coupling explicitly, indicating which resources should be consumed and which plugins should be applied. This approach supports the automation of tasks involving multiple projects and enhances dependency traceability.

Furthermore, the language enables the specification of the interfaces exposed by the microservice through the `controllers` section. In this structure, controllers, their respective paths, and available endpoints are defined. Each endpoint may specify its type (e.g., REST), name, HTTP method (such as GET or POST), and associated path, enabling the automated generation of interface-layer components.

In general, RI was designed to be expressive and extensible, covering both structural and behavioral aspects of microservices.

3.2 RI Language Grammar

The RI language is defined through a formal grammar that specifies its syntactic structure, enabling interpretation through lexical and syntactic analysis techniques. Listing 2 presents a simplified version of the language grammar.

Listing 2 RI Grammar.

```
project      : "project" STRING "{" project_statements "}"

project_statements
             : project_statements project_statement
             | project_statement

project_statement
             : use_stmt
             | git_stmt
             | config_stmt
             | folders_stmt
             | deployment_stmt
             | dependencies_stmt
             | controllers_stmt

use_stmt     : "use" IDENTIFIER
git_stmt     : "git" STRING
config_stmt  : "config" IDENTIFIER
folders_stmt : "folders" IDENTIFIER

deployment_stmt
             : "deployment" "{"
               "version" STRING
               "port" NUMBER
               "url" STRING
               "}"

dependencies_stmt
             : "dependencies" "[" dependency_list "]"

dependency_list
             : dependency_list dependency_def
             | dependency_def

dependency_def
             : "{" "plugin" STRING "project" STRING "}"

controllers_stmt
             : "controllers" "[" controller_list "]"
```

```
controller_list
             : controller_list controller_def
             | controller_def

controller_def
             : "{"
               "name" STRING
               "path" STRING
               endpoints_def
               "}"

endpoints_def
             : "endpoints" "[" endpoint_list "]"

endpoint_list
             : endpoint_list endpoint_def
             | endpoint_def

endpoint_def
             : "{"
               "type" IDENTIFIER
               "name" STRING
               "pattern" IDENTIFIER
               "path" STRING
               "}"
```

The presented grammar defines the syntactic structure of the RI language, establishing how the different elements of a project must be organized. The entry point is the `project` block, which encapsulates all other declarations.

The language interpretation process is carried out in two main stages: lexical analysis and syntactic analysis. During lexical analysis, the source code is converted into a sequence of tokens, such as keywords (`project`, `use`, `deployment`), identifiers, and literal values. Subsequently, syntactic analysis uses these tokens to construct a structured representation of the project, ensuring that the input conforms to the rules defined by the grammar [1].

Based on this structured representation, the TSE tool is able to manipulate model elements, enabling the automated generation and modification of software artifacts. In this way, the formal definition of the grammar serves as the foundation for the standardization and extensibility of the RI language.

3.3 TSE (Toolbox Service Executor)

The Toolbox Service Executor (TSE) is the tool responsible for interpreting and executing the definitions specified in the RI language. Based directly on the language lexer and parser, TSE associates semantic actions with the recognized grammar structures, enabling declarative specifications to be transformed into concrete operations over software artifacts.

Operating through a command-line interface, TSE allows project instantiation from `.ri` files as well as the execution of plugins over these projects. When processing a file, the tool performs lexical and syntactic analysis, generating a structured representation that serves as context for subsequent executions.

Once instantiated, a project is stored in the tool's internal repository and can be used as a reference in future executions. When a plugin is invoked, TSE retrieves the corresponding project context, processes the RI definition, and provides the resulting structured information to the plugin, as illustrated in Figure 1.

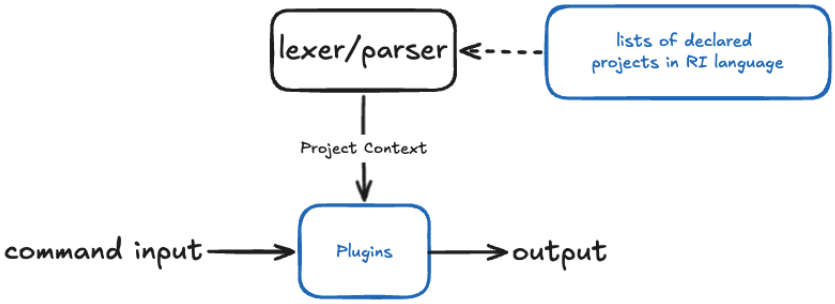

Figure 1 Simplified TSE workflow.

Plugins constitute the main extension mechanism of TSE and are defined through standardized contracts and interfaces. In general, each plugin receives a structured context derived from the RI specification, along with user-provided parameters, and produces a set of actions to be executed.

These actions are not applied directly by the plugin itself but are mediated by a standardized result structure that encapsulates operations such as file creation, content modification, and directory organization. This approach ensures greater control over execution, promoting predictability, consistency, and isolation between different plugins.

Interaction with the tool occurs through commands that specify the plugin to be executed and, optionally, the target project context. This model enables different plugins to operate over the same declarative base, supporting consistent and context-aware automation.

The TSE command-line interface is defined in Listing 3.

Listing 3 TSE command-line interface.

```
Usage:
   tse [-l] [--help]
   tse <input_file.dsl>
   tse <plugin_name> [-p] [-o <output_path>] [-f <input_file>]
   tse <plugin_name> <project_name> [-p] [-o <output_path>]

Flags:
   -p          Preview of plugin execution
   -o          Output directory path
   -f          Target file path for plugin-specific use
   -l          List all available plugins
   --help      Display help information
```

4 Case Study

This section presents a simplified example of the proposed approach, considering the existence of a previously defined service and a set of available plugins in the platform.

4.1 Available Plugins in TSE

The TSE tool provides a set of extensible plugins designed to operate over projects defined in the RI language. In this case study, two main plugins are highlighted:

- **NestJsService**: responsible for generating HTTP communication modules based on NestJS from a declarative microservice definition. The plugin interprets controllers and endpoints defined in the context and automatically generates typed services for external consumption.
- **EnvDeployment**: responsible for extracting deployment information from a microservice and injecting it into `.env`-type configuration files, enabling standardization of environment variables across different consumer projects.

These plugins are maintained within the TSE ecosystem and operate over projects instantiated through RI definitions, making them available for automated processing and integration tasks.

4.2 Initial Context

Consider a scenario in which a microservice named **ms-users**, responsible for user management, already exists within the organization. This service was developed by a separate team using Python with FastAPI, and its structure was defined in a file named *ms-users.ri* using the RI language and subsequently instantiated through the TSE tool.

```
tse ms-users.ri
```

After being processed through the RI lexical and syntactic analysis pipeline, the resulting definition is stored in the TSE repository. It can later be extended with new declarations or used as input for plugin execution.

The RI file content is shown in Listing 4:

Listing 4 User microservice project declared in RI.

```
project "ms-users" {
  use FastAPI
  git "https://github.com/user/ms-users.git"
  config default
  folders clean_architecture
  deployment {
    version "1.0.0"
    port 3001
    url "http://localhost"
  }
  dependencies []
  controllers [
    {
      name "UserController"
      path "/users"
      endpoints [
        {
          type REST
          name "getUserDataById"
          pattern GET
          path "/:id"
        }
      ]
    }
  ]
}
```

In parallel, a new team starts the development of a microservice named **ms-reports**, responsible for report generation. This service is implemented in TypeScript using NestJS and is also initially described using the RI language, as shown in Listing 5:

Listing 5 Reports microservice project declared in RI.

```
project "ms-reports" {
  use NestJS
  git "https://github.com/user/ms-reports.git"
  config default
  folders clean_architecture
  deployment {
    version "0.0.1"
    port 3000
    url "http://localhost"
  }
}
```

During development, a requirement emerges for **ms-reports** to consume data from the **ms-users** microservice. In a traditional approach, this integration would require manual implementation of HTTP clients, definition of communication contracts, and configuration of environment variables. Additionally, differences in technology stacks – such as programming languages and frameworks – often lead to inconsistent integration strategies across teams.

4.3 Scenario for Automatic Integration and Configuration Generation

In the proposed approach, these challenges are mitigated by using the RI language as a declarative source capable of being interpreted by the tool. From the analysis of the defined structures, the tool is able to infer relevant information such as endpoints, routes, and deployment configurations, making them available to plugins that automate the generation of the necessary artifacts for service integration.

Following the example, the development team responsible for **ms-reports** executes the following command:

```
tse NestJsService ms-users
```

The tool interprets the declarative definition of the **ms-users** project and automatically generates the artifacts required for its integration, including communication modules, route definitions, and standardized access structures.

As a result of this process, the plugin creates an organized directory and file structure, as shown in Listing 6:

Listing 6 Directory structure and files automatically generated.

```
user-api-service/
| dto/
| |_ index.ts
|- user-api.module.ts
|_ user-api.service.ts
```

As a result, the *UserApiServiceModule* is generated, encapsulating all HTTP communication with the ms-users microservice. The `dto` directory centralizes data transfer objects, while the `module` and `service` files represent, respectively, the module configuration and the service integration logic associated with the user service.

One of the main generated artifacts is the *user-api.service.ts*, which implements the actual calls to the service endpoints. This file is automatically generated from the RI definition, without requiring manual modifications. An example of its content is shown in Listing 7.

Listing 7 Contents of the user-api.service.ts file.

```
import { HttpService } from "@nestjs/axios";
import { BadRequestException, Injectable } from "@nestjs/common";
import { ConfigService } from "@nestjs/config";
import { firstValueFrom } from "rxjs";

@Injectable()
export class UserApiService {
  private readonly baseUrl;
  private readonly url;

  constructor(
    private readonly httpService: HttpService,
    private readonly configService: ConfigService,
  ) {
    this.baseUrl = this.configService.get<string>("MS_USERS_BASE_URL");
    this.url = `${this.baseUrl}/users`;
  }

  async getUserById(id: string): Promise<any> {
    try {
      const response = await firstValueFrom(
        this.httpService.get(`${this.url}/${id}`),
      );
      return response.data;
    } catch (error) {
      throw new BadRequestException(error, "Error at getUserById");
    }
  }
}
```

In addition, the **EnvDeployment** plugin can be used to automatically inject service deployment configurations into the consumer project environment, further eliminating the need for manual configuration.

The **EnvDeployment** plugin can be executed using the following syntax:

```
tse EnvDeployment ms-users -f .env.local
```

In this case, the plugin automatically inserts the deployment information of the selected microservice into the environment file, ensuring standardization across consumer systems.

An example of the updated `.env.local` file is shown in Listing 8:

Listing 8 Content of the .env.local file.

```
MS_REPORTS_DEPLOYMENT_VERSION=0.0.1
MS_REPORTS_DEPLOYMENT_PORT=3000
MS_REPORTS_DEPLOYMENT_URL=http://localhost

S3_ENDPOINT=************
S3_ACCESS_KEY=************
S3_SECRET_KEY=************
S3_PUBLIC_URL=************
S3_APP_BUCKET=************
S3_REGION=************

RESEND_API_KEY='************'
```

```
SYSTEM_EMAIL_ADDRESS='********* <example@gmail.com>'

# BEGIN MS_USERS_DEPLOYMENT
MS_USERS_DEPLOYMENT_VERSION=1.0.0
MS_USERS_DEPLOYMENT_PORT=3001
MS_USERS_DEPLOYMENT_URL=http://localhost
# END MS_USERS_DEPLOYMENT
```

It can be observed that the file now contains a clearly delimited section automatically generated by the plugin, containing the deployment information of the **ms-users** microservice. If these values are updated in the original service definition, the plugin can be re-executed in an idempotent manner, updating only the marked section while preserving the rest of the file.

4.4 Case Study Conclusion

The presented scenario demonstrates how the use of the RI language, combined with plugin execution in TSE, significantly simplifies the integration process between software projects. In the described example, the team responsible for **ms-reports** was able to establish communication with **ms-users** without manually implementing HTTP clients, data contracts, or environment configurations.

From the declarative definition of the existing service, the tool was able to automatically infer the required information and generate both integration modules and the associated configuration artifacts. This process reduces development effort and helps avoid inconsistencies commonly found between teams, especially in environments involving different technologies and implementation standards.

Furthermore, the example illustrates how the approach promotes the reuse of definitions and consistency across services. If changes occur in the **ms-users** specification, the corresponding artifacts can be automatically updated, keeping consumer services aligned without requiring manual rework.

This case study demonstrates, in a practical scenario, how the combination of a declarative DSL with extension mechanisms based on semantic actions can support standardization, automation, configuration management, and integration in microservice-based architectures.

5 Conclusion and Future Work

This work presented the RI language and the TSE tool as an approach for declarative specification and automated artifact generation in microservice-based architectures. Through a set of heterogeneous microservices, it was demonstrated how the platform enables project instantiation, as well as the automatic generation of integration modules and operational configurations through extensible plugins.

As directions for future work, the first and most immediate step involves validating the proposed approach in real industrial environments, allowing the RI language and the TSE platform to be evaluated under realistic development and maintenance scenarios. Although the current evaluation was conducted as a proof of concept, the RI language has already been informally presented to members of the first author's development team, where the general idea was positively received and tested through small-scale experiments. These initial interactions indicate potential practical applicability, particularly regarding standardization and automation of repetitive configuration tasks. Nevertheless, a longer evaluation period in real projects is still necessary in order to assess usability aspects, identify limitations, and observe how the approach behaves during the continuous evolution of distributed systems.

In parallel, there is also a clear need to increase the expressiveness of the RI language, enabling more detailed modeling of architectural and operational aspects of microservices. These extensions include explicit definitions for databases, persistence strategies, Docker container configurations, network policies, and service-to-service communication, as well as richer controller specifications, including formal input/output contracts and strongly typed service interfaces.

Additionally, a relevant extension of this approach is the implementation of reverse-engineering plugins, capable of analyzing existing projects and automatically converting code artifacts into RI language definitions. In this scenario, previously implemented structures such as controllers, services, and modules could be interpreted and transformed into equivalent declarative representations, significantly reducing the effort required for migration and adoption in legacy systems.

This reverse-engineering mechanism would allow the RI language to be used not only as a starting point for system generation, but also as a unifying model for representing existing software architectures. In this sense, the TSE tool would evolve into a bidirectional transformation environment, supporting both the generation and extraction of architectural definitions. This capability significantly increases the potential for adoption, especially in industrial contexts where legacy systems are a common barrier to introducing new architectures and development standards.

Within the TSE ecosystem, there is also significant potential for expanding the plugin infrastructure, since its extensible architecture enables new automations without modifying the core system. This opens space for continuous community contributions, enabling plugins for infrastructure-as-code generation, cloud provider integration, observability, automated testing, and deployment orchestration.

Therefore, the combination of an extensible DSL and a plugin-based execution platform positions the proposed approach as an environment open to incremental evolution, both at the language level and at the level of software engineering automation.

References

1 Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

2 Vaastav Anand, Deepak Garg, Antoine Kaufmann, and Jonathan Mace. Blueprint: A toolchain for highly-reconfigurable microservices. *Association for Computing Machinery*, pages 482–497, October 2023. doi:10.1145/3600006.3613138.

3 David Beazley. Ply (python lex-yacc), 2025. [Online]. Available: https://www.dabeaz.com/ply/ply.html. [Accessed: Jan. 8, 2026].

4 Morgan Bruce and Paulo A. Pereira. *Microservices in Action*. Manning Publications Co., 2019.

5 Jiajun Chen, Jiyi Wu, Yuan Zuo, Luwei Fu, and Zhiwei Zhao. Mar-dsl: A domain-specific language for iot systems implementation. In *2024 International Conference on Ubiquitous Computing and Communications (IUCC)*, pages 56–62, 2024. doi:10.1109/IUCC65928.2024.00025.

6 Thomas Erl. *SOA: Principles of Service Design*. Prentice Hall, 2007.

7 Martin Fowler. *Domain-Specific Languages*. Pearson Education, 2010.

8 David Gonzalez. *Developing Microservices with Node.JS*. Packt Publishing Birmingham, UK, 2016.

9 Pooyan Jamshidi, Claus Pahl, Nabor Mendonça, James Lewis, and Stefan Tilkov. Microservices: The journey so far and challenges ahead. *IEEE Software*, 35:24–35, May 2018. doi:10.1109/MS.2018.2141039.

10 Sam Newman. *Building Microservices*. O'Reilly Media, Inc., 2015.

11 Mark Richards and Neal Ford. *Fundamentals of Software Architecture*. O'Reilly Media, 2020.

12 Wilson Valdez Solis, Priscila Cedillo, and Attila Kertesz. Micaal: A domain-specific language for microservices in ambient assisted living. *IEEE Access*, 13:56255–56272, 2025. doi:10.1109/ACCESS.2025.3555831.

13 Alen Suljkanovic, Branko Milosavljevic, Vladimir Indic, and Igor Dejanovic. Developing microservice-based applications using the silvera domain-specific language. *Applied Sciences*, 12(13):6679, 2022.