applied sciences

MDPI

Article

Developing Microservice-Based Applications Using the Silvera Domain-Specific Language

Alen Suljkanović [1,*], Branko Milosavljević [2], Vladimir Inđić [2] and Igor Dejanović [2,*]

[1] Typhoon HIL, 21000 Novi Sad, Serbia
[2] Faculty of Technical Sciences, University of Novi Sad, 21000 Novi Sad, Serbia; mbranko@uns.ac.rs (B.M.); vladaindjic@uns.ac.rs (V.I.)
* Correspondence: alen.suljkanovic@typhoon-hil.com (A.S); igord@uns.ac.rs (I.D.)

Abstract: Microservice Architecture (MSA) is a rising trend in software architecture design. Applications based on MSA are distributed applications whose components are microservices. MSA has already been adopted with great success by numerous companies, and a significant number of published papers discuss its advantages. However, the results of recent studies show that there are several important challenges in the adoption of microservices such as finding the right decomposition approach, heterogeneous technology stacks, lack of relevant skills, out-of-date documentation, etc. In this paper, we present Silvera, a Domain-Specific Language (DSL), and a compiler for accelerating the development of microservices. Silvera is a declarative language that allows users to model the architecture of microservice-based systems. It is designed so that it can be used both by inexperienced and experienced developers. The following characteristics distinguish Silvera from similar tools: (i) lightweight and editor-agnostic language, (ii) built with heterogeneity in mind, (iii) uses microservice-tailored metrics to evaluate the architecture of the designed system, and (iv) automatically generates the documentation. Silvera's retargetable compiler transforms models into runnable code and produces the documentation for each microservice in the model. The compiler can produce code for any programming language or framework since code generators are registered as plugins. We present a case study that illustrates the use of Silvera and also discuss some current limitations and development directions. To evaluate Silvera, we conducted a survey based on A Framework for Qualitative Assessment of DSLs (FQAD), where we focused on the following DSL characteristics: functional suitability, usability, reliability, productivity, extendability, and expressiveness. Overall, the survey results show that Silvera satisfies these characteristics.

Keywords: domain-specific languages; microservice architecture; model-driven engineering; software architecture

check for updates

Citation: Suljkanović, A.; Milosavljević, B.; Inđić, V.; Dejanović, I. Developing Microservice-Based Applications Using the Silvera Domain-Specific Language. *Appl. Sci.* **2022**, *12*, 6679. https://doi.org/10.3390/app12136679

Academic Editor: Arcangelo Castiglione

Received: 2 June 2022
Accepted: 29 June 2022
Published: 1 July 2022

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Microservice Architecture (MSA) is a rising trend emerging from the enterprise world. This architectural style is defined by Lewis and Fowler [1] as "an approach to developing a single application as a suite of small services, each running in its own process and communicating using lightweight mechanisms, often an HTTP resource API".

MSAs are particularly suitable for cloud infrastructures as they greatly benefit from the elasticity and rapid provisioning of resources [2], but also for modernization of legacy systems [3].

According to the survey published by the Eclipse Foundation, in 2018 (2018 Jakarta EE Developer Survey Report—https://jakarta.ee/documents/insights/2018-jakarta-ee-developer-survey.pdf (accessed on 13 October 2021)), about 46% of organizations developed their applications in the form of microservices. A survey performed in 2019 (2019 Jakarta EE Developer Survey Report—https://jakarta.ee/documents/insights/2019-jakarta-ee-developer-survey.pdf (accessed on 13 October 2021)), again by the Eclipse Foundation, shows that MSA is the leading architecture for implementing Java in the cloud.

Although evidence shows an increase of distributed systems developed in the form of MSA, studies presented in [3–5] show that it is still problematic to find skilled developers familiar with microservices. Microservice developers need to familiarize themselves with the variety of technologies and tools in a timely manner. Furthermore, supporting polyglot programming may require additional tooling, processes, and knowledge [6]. Being polyglot is an important characteristic of MSA as it gives developers the flexibility to choose the technology stack and languages that work best for their needs. However, introducing many languages and frameworks may actually decrease the overall understandability and maintainability of the system. As shown by Wang et al. [7], some organizations already regulate language diversity by restricting the number of programming languages used in a microservice-based system to a few core languages. While this decreases the problems of understandability and maintainability, a study performed by Baškarada et al. [4] shows that the opportunity to use heterogeneous technology stacks was singled out by most interviewees as one of the most significant drivers for MSA adoption.

Due to the higher complexity of MSA, migration of the monolithic legacy system to MSA can cause high development costs. The empirical study performed by Lenarduzzi et al. [8] shows that the technical debt of microservices grows 90% slower than in the corresponding monolithic legacy system. Lenarduzzi et al. [8] also recommend companies define a set of service templates as a way to ease the development of new microservices, but also not to delay important architectural decision making, as this will cause more effort in the future. Having service templates helps, but developers are still required to manually introduce the necessary changes to create a new microservice or to update the existing microservices to comply with architectural changes.

Documenting MSA properly can also be a challenge. Frequent changes in the architecture often lead to wrong and out-of-date architecture models [9]. In addition, communication relationships between microservices and the specific APIs through which they communicate are often missing from such models [9]. With the highly decentralized development and design of microservices, it becomes challenging to maintain a centralized reference of the architectural design [10,11]. Due to this, the system's architecture deviates from the original design over time.

The main contributions of this paper are as follows:

- We present *Silvera*, a Domain-Specific Language (DSL) (DSLs are expressive languages tailored for a specific domain [12] (see Section 2.3)) and an extensible code generator framework that aims to accelerate the development of microservices.
- We present a case study that shows that DSLs significantly accelerate the development of MSA (Section 5.7.1).
- We also provide a description of the development process (Section 3.1) and the study to assess the language based on quality characteristics defined by A Framework for Qualitative Assessment of DSLs (FQAD) [13].

Silvera aims to satisfy the following: (i) the language is easy to use for both domain experts and beginners, (ii) it supports well-known MSA design patterns as first-class concepts, (iii) it supports heterogeneous technology stacks through an extensible code generators framework, (iv) it provides automatic generation of architecture diagrams and OpenAPI documentation, and (v) it uses microservice-tailored metrics to evaluate the architecture of the designed system.

To assess the feature set of Silvera, we compare it to similar available tools and frameworks in Section 2.4. The development process of Silvera, design decisions, used development patterns, and the language abstract and concrete syntaxes are described in Section 3. Silvera is based on well-established principles of Model-Driven Engineering (MDE) which promote the active use of models throughout the software development process, leading to an automated generation of the final application. The models can be defined using general-purpose modeling languages (such as the UML), but DSLs are often used for restricted, well-known domains. Users can use Silvera to *model* microservice-based systems and *automatically* translate models into working applications. The transformation

of the specification to a runnable code is performed by the compiler. The compiler supports an arbitrary number of code generators where each code generator produces code for a corresponding programming language and/or framework.

To better explain the language usage, we investigate building a small MSA application in Section 4 that serves as a short demonstration of Silvera's capabilities. Even though small, the application is complete and demonstrates the implementation of almost all design patterns supported in Silvera. The application model contains definitions of microservices, their respective APIs, domain models, and description of inter-service communication.

Furthermore, to evaluate Silvera's quality characteristics, we conducted a survey based on FQAD where we focused on the following DSL characteristics: functional suitability, usability, reliability, productivity, extendability, and expressiveness. The results of the study and its limitation are discussed in Sections 5–7.

Silvera is a free and open-source project, and it is hosted on GitHub (Silvera project—https://github.com/alensuljkanovic/silvera (accessed on 1 June 2022)).

The remainder of the paper is organized as follows. Section 2 presents related work. Section 3 presents the implemented language, whereas Section 4 shows how Silvera can be used during the implementation of microservice-based applications. Section 5 presents results of the evaluation of Silvera language. Threats to validity are shown in Section 6. Section 7 discussed the limitations of the current approach and implementation. Section 8 concludes the paper and discusses future improvements of Silvera.

2. Related Work

2.1. Comparison of MSA with Other Architectural Styles

The terms *architecture* and *architecture description* are introduced by ISO/IEC/IEEE 42010:2011 standard [14]. An *architecture description* expresses an architecture of a system-of-interest [14]. *Architecture* encompasses fundamental concepts or properties of a system in its environment embodied in its elements and relationships and in the principles of its design and evolution [14]. Architecture descriptions are used by software teams to improve communication and cooperation among stakeholders, enabling them to work in a comprehended and coherent manner. An architecture description comprises architecture views and models. Gorski [15] presents the software architecture model 1+5 that encompasses various architectural views for modeling business processes, describing use cases and their realizations, interaction and contract agreements between services, and deployment.

MSA is a *service-based architecture*, just like Service-Oriented Architecture (SOA). Even though MSA and SOA represent very different architectural styles, they share many characteristics. Services are a primary architecture component used to implement and perform business and nonbusiness functionality in both MSA and SOA [16]. Both MSA and SOA are generally distributed architectures and also lend themselves to more loosely coupled and modular applications [16]. Furthermore, the implementation of a service is hidden behind its publicly available API. Due to this, the implementation of a service can be entirely changed without affecting the rest of the system as long as the API changes are backward compatible [16].

Although MSA and SOA both rely on services as the main architecture component, they vary greatly in terms of service characteristics [16]. Differences are shown in service taxonomy (i.e., how services are classified within an architecture), service ownership, and service granularity.

As shown in Table 1, microservices have limited service taxonomy. There are only two service types: *functional services* and *infrastructure services*. Functional services implement specific business operations or functions, whereas infrastructure services implement non-functional tasks such as authentication, authorization, auditing, logging, and monitoring. In MSA, infrastructure services are not exposed to the outside world and are only available internally to other services [16]. On the other hand, in SOA, there are four types of services. *Business services* are abstract, coarse-grained services that define core business operations. These services are devoid of implementation details and usually only contain information about a service name and expected inputs and outputs. *Enterprise services* are concrete,

coarse-grained services that implement the functionality defined by the business services. These services are generalized and shared across the organization [16]. *Enterprise services* can contain business functionality, but usually, they rely on application and infrastructure services. *Application services* are fine-grained services that are bound to a specific application context. They provide functionality not found in the enterprise services. *Infrastructure* services implement the same tasks as in MSA. In addition, there is a significant difference in service ownership as development teams are responsible for full support and development of a service throughout its life cycle (also known as the "you build, you run it" principle) [17]. Microservices are small, fine-grained services (hence "micro"), whereas services in SOA range in size from very small to large enterprise services.

Table 1. Comparison of service characteristics in MSA and SOA.

Architectural Style	Service Taxonomy	Service Ownership
MSA	Functional services	Application development teams
	Infrastructure services	Application development teams
SOA	Business services	Business users.
	Enterprise services	Shared services team or architects.
	Application services	Application development teams.
	Infrastructure services	Application development teams or infrastructure services teams.

MSA and SOA also differ with regard to data sharing and service coordination. In the case of data sharing, MSA promotes a style where microservices share as little data as possible, whereas SOA promotes the diametrically opposed concept of sharing as much data as possible [16]. For this reason, a microservice and its associated data represent a single unit with minimal dependencies, which facilitates maintenance and deployment of the microservice (i.e., microservice can be changed and redeployed without affecting the rest of the system). Multiple services can be composed together as a new service. This process is known as *service composition*. Service composition implies the existence of *service coordination*. For service coordination, MSA focuses on service choreography, whereas SOA relies on both service choreography and service orchestration [16]. The term *service orchestration* refers to the coordination of multiple services through a centralized mediator, whereas *service choreography* lacks the mediator [16]. Implementation of service coordination is a non-trivial and error-prone task. However, there are several tools that allow developers to automate this process [18,19]. As a result of this approach, microservices are responsible for interaction with others [20]. Of course, one can still utilize orchestration; however, this is not a typical approach [20]. Due to the mentioned differences, systems built on SOA tend to be slower than microservices and require more time and effort to develop, test, deploy, and maintain [16]. In addition, a service composition should be efficient with minimal execution time and energy consumption. An approach for execution time and energy efficient service composition is presented by Li et al. in [21]. The order of service tasks execution is determined by a *service scheduling* [22]. Resource allocation is another important process that affects the performance. The goal of resource allocation is to select resources for component instances and determine the number of component instances needed to meet performance and reliability requirements [22]. Usually, microservices have more components with less functionality that require fewer resources.

MSA also differs from Monolithic Architecture (MA). In MA, components rely on the sharing of resources of the same machine (memory, databases, or files) and are therefore not independently executable [23]. Monolithic applications are usually internally split into multiple services and/or components, but they are all deployed as a single solution. Unless the application is becoming too big, monolithic applications are easier to develop [24]. However, large monolithic applications suffer from the following problems [17]: they are difficult to maintain and evolve, it is hard to add or update libraries due to *dependency hell*, change in only one module requires rebooting the whole application, etc. Microservices

succeed in mitigating these problems and are gaining in popularity due to the following characteristics:

- *Size*: Because microservices implement a limited amount of functionalities, their code bases are small which limits the scope of a bug [17]. The small size also provides benefits in terms of service maintainability and extendability. A small service can be easily modified or rebuilt from scratch with limited resources and in limited time [25].
- *Independence*: Each microservice in MSA is operationally independent of others and communicates with other microservices through their published interfaces [25]. This has several benefits: (i) microservices are independently deployed, (ii) the *new* version of microservice can co-exist with the *old* version, and the microservices that use the *old* microservice can be gradually modified to use the *new* microservice [17], (iii) changes in one microservice do not require the reboot of the whole system, and (iv) scaling MSA implies deploying or disposing of instances of microservices with respect to their load [26].

2.2. MSA Design Patterns

In this section, we define design patterns that will be used in the rest of the paper.

A design pattern is a reusable solution for a problem that occurs in a specific context. Richardson [27] proposes a taxonomy and describes design patterns used in MSA. In the rest of the paper, design patterns from the following categories will be considered: *Deployment* patterns, *Communication* patterns, *External API* patterns, *Reliability* patterns, and *Service Discovery* patterns. Each of the previously mentioned categories can contain multiple different design patterns, as shown in Table 2.

Deployment patterns provide a solution to the problem of packaging and deploying microservices. Patterns such as *Single Instance Per Host*, *Multiple Instances Per Host*, *Single Instance Per Container*, and *Serverless* are all deployment patterns. In the case of the *Single Instance Per Host* and the *Multiple Instances Per Host*, a *Host* can be either a physical or a virtual machine (VM). The most popular deployment pattern, both in industry and academia, is the *Single Instance Per Container*, followed by the *Single Instance Per Host*, where a host is a VM [28]. There are several reasons why containers are preferred over VMs [28]: (i) creating and launching container images are often very fast, (ii) the same physical server can hold more containers than VMs due to their size, and (iii) more than one container can use a single operating system, which in turn reduces the overhead of licensing costs compared to the VM.

Microservices must handle requests from external clients, but often they must collaborate to perform a certain task. The role of *Communication* patterns is to provide a solution for a problem of communication between the parts of the system. The most prevalent communication patterns in microservice-based systems are *Remote Procedure Call* (RPC) and *Messaging*. With RPC, microservices can communicate either in a synchronous or asynchronous manner, whereas with *Messaging* the communication is always asynchronous. According to Aksakalli et al. [28], it is not possible to determine the most popular communication pattern, since the selected pattern can change over time as the system evolves.

How microservices will establish a connection is further described by *Service Discovery* patterns. The *Service Registry* pattern ensures the existence of a service registry, which represents a database that contains the data model of available microservices deployed in the platform [29]. A service registry is a critical part of the system and must be highly available. The responsibility for registering or unregistering a microservice within a service registry can be put either on the microservice itself (*Self-Registration*) pattern or the third party (*Third-Party*) pattern. Similarly, the responsibility for discovering a microservice can be either on the client side (*Client-Side Discover*) pattern or on the server side (*Server-Side Discovery*) pattern.

Table 2. Microservice design patterns. Descriptions of all presented designed patterns are given by Richardson [27].

Purpose	Name	Description
Deployment	Single instance per host	Each service instance will be deployed to a separate host.
	Multiple instances per host	Multiple service instances deployed to a single host.
	Single instance per container	Each service instance will be deployed to a separate container.
	Serverless	Service instances are run by the deployment infrastructure that hides any concept of servers.
Communication	Remote Procedure call	Services use RPC-based protocol for communication.
	Messaging	Services communicate in asynchronous manner, via passing messages.
External API	API gateway	Provides a single entry point for all external clients.
	Back-end for front-end	Provides a single entry point for each client separately.
Reliability	Circuit breaker	Prevents a network or service failure from cascading to other services.
Service discovery	Client-side discovery	Client service obtains the location of the server service.
	Server-side discovery	Client services' send requests to the server service via the router.
	Service registry	Contains service instances and their locations.

External API patterns describe exactly how external clients will communicate with microservices. An *API Gateway* provides a single entry point for all external clients. To avoid a single point of failure, multiple instances of an API gateway are usually deployed [28]. Additionally, a separate API gateway can be provided for each kind of client (web app, mobile, etc.). This variation of an *API Gateway* pattern is a *Back-End for Front-End* pattern.

Microservices are *built to fail* [30]. The solution for the problem of preventing the microservice failure to cascade to other microservices is provided by a *Reliability* pattern named *Circuit Breaker*.

2.3. Domain-Specific Languages

Van Deursen et al. [31] define DSL as *a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain*. DSL does not attempt to address all types of computational problems or not even large classes of such problems [32]. This allows the language to be very expressive for problems that fall in the domain. DSLs foster the building of a community of domain experts who *speak the same language* [33]. DSLs whose domain, abstractions, and notations are closely aligned with how domain experts (i.e., non-programmers) express themselves allow domain experts to easily read and often write program code since it is not cluttered with irrelevant implementation details [12]. The level of involvement of domain experts may vary. However, it is important for the DSL to provide a syntax that can be read and comprehended by domain experts even if they do not type the program directly. A DSL that domain experts can read can serve as a design and implementation tool and as a requirements elicitation medium [34]. The empirical

studies suggest that the use of DSLs increases flexibility, productivity, reliability, and usability [35,36].

The study performed by Johanson and Hasselbring [37] shows that domain experts achieve significantly higher accuracy and spend less time solving tasks when using their DSL instead of the comparable GPL-based solution. A study performed by Kosar et al. [38] shows that participants are more effective and efficient with DSLs than with GPLs, especially in domains where participants were less experienced. The study also suggests that in cases where a DSL is available, developers will perform better when using the DSL than using a GPL. In this study, we did not explicitly compare our DSL with GPLs, but we showed that the use of DSLs gives good results compared to the direct use of the technologies with which respondents were experienced.

2.4. Existing MSA Frameworks

In this section, we will briefly describe current state-of-the-art tools for describing the architecture of distributed systems based on microservices.

MAGMA [39], AjiL [40], TheArchitect [41], Microbuilder [42], JHipster (JHipster—https://www.jhipster.tech/jdl/ (accessed on 14 February 2021)), LEMMA [43], and Jolie [44] are currently available tools for describing distributed systems based on microservices. The main goal of these tools is to accelerate the development of microservices. By using these tools, users can model their systems and use the model to generate runnable applications.

MAGMA (*Maven Archetype for Generating Microservice Architectures*) is a tool that is based on the Maven build management system. It aims at accelerating the development of microservices architectures (MSA) by generating infrastructure code that is: (i) specifically configured for the target application domain; (ii) directly runnable; and (iii) extensible with user-defined templates [39]. Users can easily create their systems through a simple input wizard. Services such as *discovery service*, *security service*, *user management service*, and *resource service* come pre-implemented in MAGMA and are all available in the input wizard. The input wizard contains a small preview panel, which shows the architecture of the system to be generated. Although it is easy to use, it can be troublesome and time-consuming to setup MAGMA, especially for developers not accustomed to Maven. In addition, since MAGMA generates only project shells for services defined through user-defined templates, a domain model, together with an API and business logic, must be implemented manually.

AjiL is a tool for creating and describing MSAs based on the Eclipse Modeling Framework (EMF) (Eclipse Modeling Framework—https://www.eclipse.org/modeling/emf/ (accessed on 10 January 2021)). AjiL comprises a graphical editor and a template-based code generator. The generator converts existing AjiL diagrams into runnable MSA, based upon Java and the Spring framework [40]. AjiL's graphical editor is nicely designed and easy to use. Since AjiL comes with the same set of pre-implemented services as MAGMA, modeling of MSA can be performed quickly. However, generating code from the model is not straightforward. To generate code from a model, the model needs to be copied into the generator project because there is no connection between the graphical editor and the code generator. In addition, AjiL currently provides no mechanism for the integration of manually added code with generated code.

TheArchitect is a rule-based system used for generating serverless microservices [41]. MSAs in TheArchitect are created by providing system requirements into TheArchitect's input wizard. The obtained information is processed by the predefined set of models afterward. These models are subsequently analyzed by the code generator, which produces the application code.

MicroBuilder is a tool for generating REST-based (REST–Representational State Transfer) microservices [42]. It consists of two modules: MicroDSL and MicroGenerator. MicroDSL is a domain-specific language (DSL) used to describe the architecture of the microservices. MicroGenerator uses the MicroDSL specifications to generate Java applications based on the Spring framework.

JHipster is a tool for generating, developing, and deploying web applications and MSAs. MSAs in JHipster are modeled by using the JHipster domain language (JDL). Modeling of microservices in JDL can be performed quickly due to its user-friendly syntax. JDL specifications are transformed automatically into runnable Java applications by the JHipster code generator. JHipster can be used online or installed locally with NPM (NPM—https://www.npmjs.com/ (accessed on 18 January 2022)), Yarn (Yarn—https://yarnpkg.com/ (accessed on 18 January 2022)), or Docker (Docker–https://www.docker.com/ (accessed on 18 January 2022)). However, the online version is quite limiting by default, and to achieve the full experience, users must allow access to either GitHub or GitLab accounts. Installing JHipster locally can be troublesome, but it is the preferred way.

LEMMA (Language Ecosystem for Modeling Microservice Architecture) is a set of Eclipse-based modeling languages and model transformations for developing MSA [43]. These languages provide different modeling viewpoints for different roles in a microservice development team [45]. By introducing explicit modeling viewpoints, LEMMA decomposes the system into smaller, more specialized parts. Because of this, each role is presented only with the information relavant for that role [45]. Just like AjiL, LEMMA is also based on the Eclipse ecosystem.

Jolie is a service-oriented programming language [44]. A service, in Jolie, is composed of two parts: behavior and deployment. A behavior part defines the implementation of the service's functionalities, whereas the deployment part defines the necessary information for establishing communication links between services [44]. The Jolie interpreter is implemented in Java, and it comes with a Java API to interact with it.

Because both TheArchitect and MicroBuilder are not publicly available, unlike the rest of presented tools (which are all open-source projects), we were unable to study them in greater detail.

The comparison of available features in Silvera, MAGMA, AjiL, JHipster, LEMMA, and Jolie is shown in Table 3, whereas a comparison of implemented microservice design patterns is shown in Table 4. Table 4 contains only patterns implemented by at least one of the tools.

Silvera is a lightweight language. Because of this, it can be used in any text editor, whereas AjiL is tightly coupled with its GUI editor. The simple textual notation also enables easier collaboration through version control systems. Silvera shares this characteristic with JDL.

All tools presented in this section, except Jolie, implement the *API Gateway* pattern from the *External API* group of patterns. Similarly, these tools also implement the *Circuit Breaker* pattern from the *Reliability* group of patterns. In Jolie, these patterns are not implemented as part of the language. However, Jolie offers composition primitives that can be used for manual implementation of these patterns.

Unlike other tools presented in this section, both Silvera and LEMMA are built with heterogeneity in mind. LEMMA provides a *Technology Modeling Language* (TML), where users model custom technology aspects. The disadvantage of TML is that it does not support versioning, meaning that only one version of the *Technology Language* is supported. On the other hand, Silvera supports an arbitrary number of programming languages and their versions (see Section 3.4). Various implementations of microservices, API gateways, service registries, and message brokers can also be supported easily. This allows developers to use the best tool for the job and also supports experimentation.

Silvera uses microservice-tailored metrics to evaluate the architecture of the designed system. Besides TheArchitect, no other tool presented in this section supports architecture evaluation. However, metrics used by TheArchitect are mainly derived from *Object-Oriented* design and, as such, are not fully applicable to microservice-based systems [46]. Additionally, in Silvera, users can easily provide custom evaluation functions (see Section 3.4.3).

Table 3. Comparison of available features in Silvera with other tools.

Tool	Textual Notation	GUI Editor	Arch. Evaluation	Database Support	Target Language
Silvera	SilveraDSL	no	yes	any	any
MAGMA	Java	yes	no	MySQL	Java
AjiL	XML	yes	no	MySQL, MongoDB	Java
JHipster	JDL	yes	no	MySQL, MongoDB, PostgreSQL, etc.	Java
LEMMA	Several DSLs	yes	no	MariaDB, MongoDB	Java, Python
Jolie	Jolie	yes	no	All supported by JDBC driver	Java, Javascript

Table 4. Comparison of microservice design patterns implemented in Silvera with other tools.

Tool	Deployment	Communication	External API	Reliability	Service Discovery
Silvera	Single instance per host, Single instance per container	Remote Procedure call, Messaging, custom	API gateway	Circuit breaker	Client-side discovery, Service registry
MAGMA	Single instance per host, Single instance per container	-	API gateway	Circuit breaker	Service registry
AjiL	Single instance per host	Remote Procedure call, Messaging	API gateway	Circuit breaker	Service registry
JHipster	Single instance per host, Single instance per container	Remote Procedure call	API gateway	Circuit breaker	Service registry
LEMMA	Single instance per host, Single instance per container	Remote Procedure call, Messaging	API gateway	Circuit breaker	Service registry
Jolie	-	Remote Procedure call, Messaging	-	-	-

Another characteristic that distinguishes Silvera from the rest of the presented tools is the simple installation procedure. Silvera comes with a small number of dependencies and is available at the *Python Package Index* (Python Package Index—https://pypi.org/ (accessed on 18 January 2022)). To install Silvera, use `pip install silvera` command (*pip* is the package installer for Python, available at https://pypi.org/project/pip/ (accessed on 18 January 2022)).

3. Silvera

In this section, we give an overview of the Silvera language.

Silvera is a declarative language developed for the domain of microservice software architecture development. We call these types of DSL "technical DSLs" or "horizontal DSLs". The language is designed in a way that directly implements design patterns related to the domain of MSA.

3.1. Implementation Phases

There are several approaches to the implementation of DSLs. These approaches were first classified by Spinellis [47] in the form of a collection of design patterns. His work was later extended by Mernik et al. [48]. The patterns are associated with the development phases of DSLs, which results in the following classes of patterns:

- *Decision patterns*—which provide a set of common situations for which DSLs have been successfully created in the past,
- *Analysis patterns*—which provide a set of guidelines on how to identify the problem domain and gather the domain knowledge,
- *Design patterns*—which provide a set of guidelines on how to design a DSL,
- *Implementation patterns*—which provide a set of guidelines on how to choose the most suitable implementation approach.

Silvera was developed in four successive phases: *Analysis*, *Decision*, *Design*, and *Implementation*.

Analysis phase. During the analysis phase, we used the analysis patterns from Table 5.

Table 5. Analysis patterns used during the domain analysis phase.

Pattern Name	Description
Informal	The domain is analyzed in an informal way [48].
Extract from code	Mining of domain knowledge from legacy GPL code by inspection or by using software tools, or a combination of both [48].

Most of the domain knowledge was gathered by analyzing the available literature, the code, and documentation (*Extract from Code*) pattern of available systems. We analyzed the domain in an informal way (*Informal*) pattern, and we gathered the literature mostly via the snowballing approach. We gathered a body of relevant papers, searched the papers that were in the reference list of these starting papers (*Backward Snowballing* [49]), and the papers that cite these starting papers (*Forward Snowballing* [49]). The output of this phase consisted of domain-specific terminology and semantics in more or less abstract form [48].

Decision phase. During this phase, we used the decision patterns from Table 6.

Table 6. Decision patterns used during the decision phase.

Pattern Name	Description
Task automation pattern	GPL programming tasks that are tedious and follow the same pattern can be generated automatically by an application generator (compiler) for an appropriate DSL [48].
AVOPT pattern	DSL makes operations such as domain-specific analysis, verification, optimization, parallelization, and transformation of application programs possible. These operations are usually not feasible in a GPL due to complexity [48].

The decision to create a new DSL stemmed from the fact that we wanted to automatically generate the infrastructure code (*Task Automation*) pattern and also to be able to perform domain-specific analysis and evaluation of the designed microservice-based system (*AVOPT*) pattern.

Design phase. This phase can be characterized along two orthogonal dimensions: the relationship between the DSL and existing languages and the formal nature of the design description [48]. During this phase, we used patterns shown in Table 7.

Table 7. Design patterns used during the design phase.

Pattern Name	Description
Language invention	A DSL is designed from scratch with no commonality with existing languages [48].
Informal	DSL is described informally [48].
Formal	DSL is described formally using an existing semantics definition method such as attribute grammars, rewrite rules, or abstract state machines [48].

The easiest way to design a DSL is to base it on the existing language. DSLs built this way are called *Internal DSLs*. The advantage of this approach is that no new language infrastructure has to be built, but the downside is the limited flexibility since a DSL has to be expressed by using concepts of the host language [50]. Another approach is to create a so-called *External DSL*. An external DSL is a completely independent language built from scratch. As external DSLs are independent of any other language, they need their own infrastructures such as parsers, linkers, compilers, or interpreters [50]. Silvera is an external DSL with no relationships with any existing languages (*Language Invention*) pattern.

Mernik et al. [48] distinguish between informal and formal designs. In an informal design, the language specification is usually in some form of natural language [48]. In a formal design, a language syntax is usually specified via regular expression and grammar, whereas a semantic is specified via attribute grammars, rewrite systems, and abstract state machines [48]. The formal design has several benefits [48]: (i) brings problems to light before the DSL is actually implemented and (ii) can be implemented automatically by language development tools, which significantly reduces the implementation effort. Silvera's syntax specification is defined in the form of PEG (*Parsing Expression Grammar*) grammar (*Formal*) pattern. However, Silvera's semantic specification is defined by code generators (*Formal*) pattern.

Implementation phase. In this phase, we considered multiple implementation patterns, as shown in Table 8.

We chose *Compiler Application Generator* pattern over other patterns, such as *Interpreter*, *Embedding*, *Extensible Compiler/ Interpreter*, and *Commercial-Off-The-Shelf* approach. A disadvantage of this approach is the higher cost of building the compiler from scratch. However, this approach also yields advantages such as closer syntax to the notation used by domain experts, good error reporting [48], and minimized user effort to write correct programs [51]. The patterns *Compiler/Application Generator* pattern and *Interpreter* offer similar advantages and disadvantages [48], but we chose the former due to execution speed.

Table 8. Implementation phase patterns considered during the implementation phase.

Pattern Name	Description
Interpreter	DSL constructs are recognized and interpreted using a standard fetch–decode–execute cycle. This approach is appropriate for languages having a dynamic character or if execution speed is not an issue [48].
Compiler/application generator	DSL constructs are translated to base language constructs and library cells. A complete static analysis can be performed on the DSL program/specification [48].
Embedding	DSL constructs are embedded in an existing GPL (the host language) by defining new abstract data types and operators [48].
Extensible compiler/ interpreter	A GPL compiler/interpreter is extended with domain-specific optimization rules and/or domain-specific code generation. While interpreters are usually relatively easy to extend, extending compilers is hard unless they were designed with extension in mind [48].
Commercial-Off-The-Shelf (COTS)	Existing tools and/or notations are applied to a specific domain [48].

3.2. Silvera Abstract Syntax

In this section, we present the abstract syntax of the Silvera language. Silvera's abstract syntax is specified in the form of a metamodel. The simplified version of the metamodel, for brevity, is presented in Figures 1 and 2.

The main concept of the Silvera metamodel is the *Model*, which consists of one or more modules (*Module*). A module enables users to logically organize their Silvera code. Each module can consist of declarations of microservices (*ServiceDecl*), API gateways (*APIGateway*), service registries (*ServiceRegistryDecl*), configuration servers (*ConfigServerDecl*), dependencies (*Dependency*), message pool (*MessagePool*), or message brokers (*MessageBroker*). Each declaration can be identified by its name which is unique at the module level. The unique identifier for a declaration within the model is its fully qualified name (FQN), which is calculated by the following formula:

$$< module\_path > . < module\_name > . < declaration\_name > \quad (1)$$

A module can reference declarations from another module by importing it.

For each microservice, it is possible to define its name, API, deployment strategy, communication style, whether the microservice should be registered within a service registry, and whether it should draw external configuration data from a configuration server.

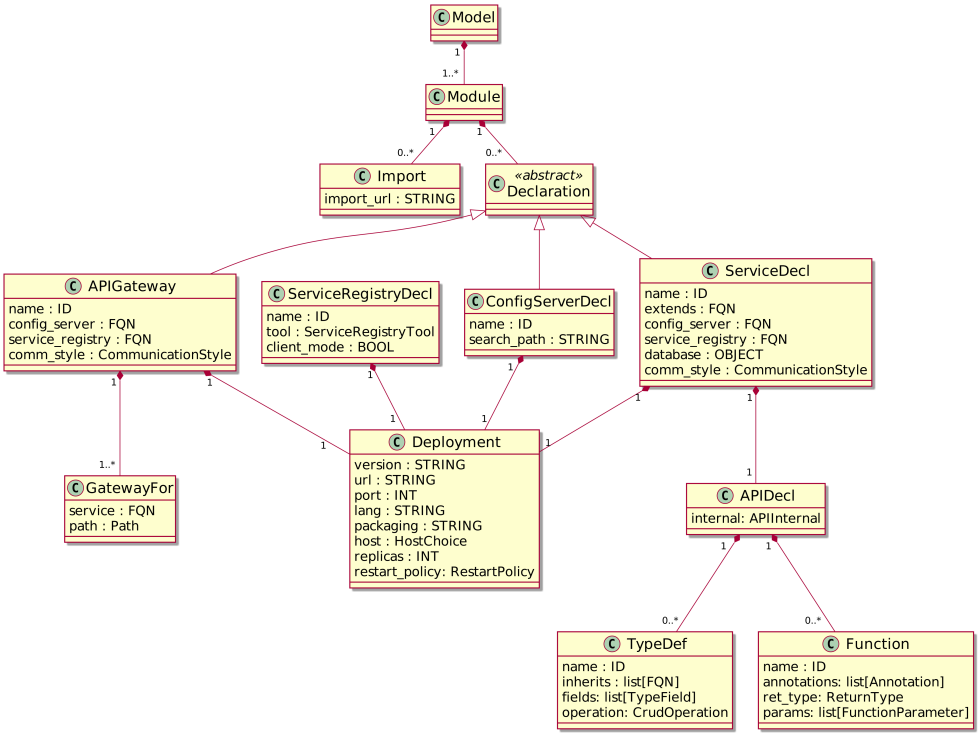

Figure 1. The simplified version of the Silvera metamodel (part 1).

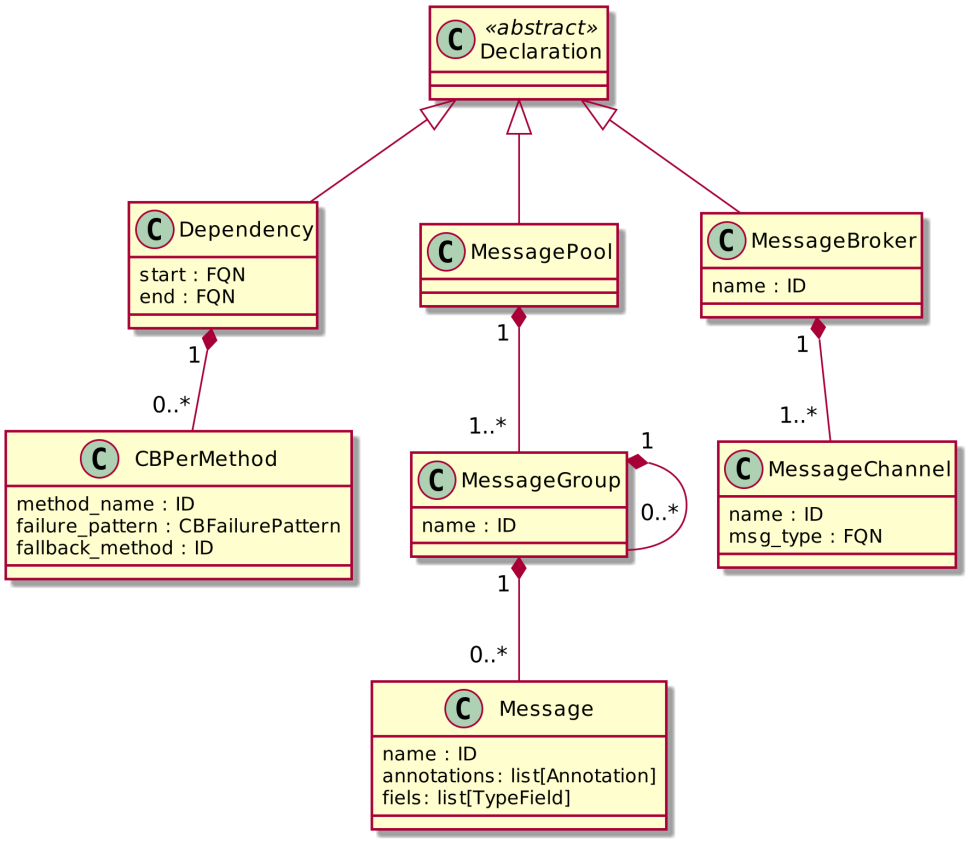

Figure 2. The simplified version of the Silvera metamodel (part 2).

The API (*APIDecl*) declaration consists of function definitions and the definition of service-specific objects used for modeling microservice business entities. Each function definition consists of a function name, function parameters, return type, and annotation (optional), whereas the definition of the service-specific object (*TypeDef*) is given by its name and one or more fields (*TypeField*). Every field has its name and a data type, but can also have a special ID attribute, which is used to identify fields during serialization and deserialization of messages in a binary format and to ensure backward compatibility for newer versions of the API. For that reason, once assigned, an ID attribute should not be changed.

In Silvera, each microservice has a particular communication style. Communication style defines a protocol used to send and receive messages. Currently, it is possible to choose between *RPC-based* and *messaging-based* communication styles. The format of messages that can be sent and received from a microservice is defined by its API. Microservices that use RPC to call methods from other microservices must define those microservices as dependencies. RPC-based communication is synchronous by default, but Silvera supports asynchronous RPC communication as well.

Since microservices can fail at any time, MSAs must be designed to cope with failures [1]. The failure of one microservice should not take down the whole system. One of the design patterns that helps in mitigating such problems is the *Circuit Breaker* pattern (see Section 2.2), which is directly supported in Silvera. Failure recovery must be defined for every API function that the *start* microservice calls from the *end* microservice. Table 9 shows failure recovery strategies supported by Silvera.

Table 9. Failure recovery strategies supported by Silvera.

Strategy Name	Description
fail_fast	An exception will be raised in the client if an API call fails (default behavior)
fail_silent	Returns an empty response
fallback_static	Returns default values
fallback_stubbed	Response is a compound object where each field has either a default value or a value determined from the request state
fallback_cache	Returns a cached version of response if present, otherwise returns an empty response such as *fail_silent*
fallback_method	Defines a method that will be called in case the original method fails

When using a messaging-based communication style, microservices communicate asynchronously (via passing messages). Every message type (*Message*) used within the system is defined in the message pool (*MessagePool*). The Message pool is globally available and can be referenced from any module. Messages are delivered to their destinations by message brokers (*MessageBroker*). Every message broker contains one or more message channels (*MessageChannel*). The message broker creates message channels, and each channel can have multiple consumers and/or producers. The consumer is every microservice that reads messages from a channel. Analogously, the producer is a microservice that puts messages inside the channel. Message channels in Silvera are typed, which means that each channel is dedicated to a specific message type. Messages can only be consumed or produced by API methods. For a microservice to send a message, its API method must be registered as a publisher to a channel whose specific purpose is to communicate that kind of message. Likewise, to receive a message of a particular type, its API method must be registered as a consumer of a channel that contains a given message type. The method can be registered as a consumer with an *@consumer* annotation and as a publisher with an *@producer* annotation. The method can, at the same time, be both consumer and producer.

Message channels are logical addresses in the messaging system; how they are actually implemented depends on the messaging system product and its implementation.

In MSA, client applications usually need to collect data from more than one microservice. If the communication is direct, the client needs to communicate with multiple microservices to collect the data. Such communication is inefficient and increases the coupling between the client and the microservices [27]. An alternative is to implement an *API Gateway*. An API gateway represents a single entry point for all clients, and it can handle requests in one of two ways: (a) requests are routed to the appropriate service, and (b) requests are fanned out to multiple microservices. In Silvera, an API gateway is a special service implemented in the form of an *APIGateway* object. Its attribute *gateway_for* determines which microservices will be put behind the gateway. In the current implementation of Silvera, the API gateway only serves as a router of requests. In the future, we plan to expand on this implementation by providing security features, implementing the *API Composition* [27] pattern and adding the option to restrict services' APIs to a certain set of operations. The *API Composition* pattern uses an API composer, or aggregator, to implement a query by invoking individual microservices that own the data and then combine the results by performing an in-memory join [27].

In Silvera, each microservice can define its specific deployment requirements. Deployment is managed by the *Deployment* object, with the following attributes: *version*, *url*, *port*, *lang*, *packaging*, *host*, *replicas*, and *restart_policy*. Attribute *version* defines a version of a microservice. Attributes *url* and *port* define a location of a microservice on a computer network. Attributes *lang* and *packaging* define a programming language in which the microservice will be implemented and in which form it will be used (source or binary). Attribute *host* defines whether a microservice will run on a physical host, virtual machine, or inside a container, whereas attribute *replicas* defines a number of instances of a microservice. Finally, the attribute *restart_policy* defines when a microservice should be restarted (after failure, always, etc.). This attribute can be currently used only if the host is a container.

A service registry is a special service that contains information about a number of instances and locations of each microservice in the system. In Silvera, the service registry is implemented in the form of *ServiceRegistryDecl* object. This object contains the following attributes: *tool*—which defines which tool will be used as a service registry, *client_mode*—which defines whether the service registry could be registered within another service registry. Since it is a special type of microservice, a service registry can also be deployed in various ways by using the *Deployment* object. The microservice is registered within the service registry by providing a reference to a *ServiceRegistryDecl* object to its *service_registry* attribute.

In Silvera, microservices can draw configuration files from an external configuration server. The configuration server is implemented in the form of a *ConfigServerDecl* object.

3.3. Silvera Concrete Syntax

In this section, we present the concrete syntax of the Silvera language. Concrete syntax defines how the abstract syntax concepts are presented to the user [12]. It is possible to create multiple concrete syntaxes (textual, graphical, etc.) for a single abstract syntax. Silvera's concrete syntax is provided in the form of textual notation. In Section 3.3.1, we have shown an excerpt from the grammar. We omitted some parts of the rules for brevity. The full version of the grammar is available on GitHub (Silvera grammar—https://github.com/alensuljkanovic/silvera/blob/master/silvera/lang/silvera.tx (accessed on 1 June 2022)).

3.3.1. Microservice Declaration

In Listing 1, we show a simplified grammar rule for a microservice declaration. Some parts are omitted for brevity.

Listing 1. An excerpt of the grammar rule for a microservice declaration.

```
1  ServiceDecl:
2    'service' name=ID '{'
3     ('config_server' '=' config_server=FQN)?
4     ('service_registry' '=' service_registry=FQN)?
5     (deployment=Deployment)?
6     'communication_style' '=' comm_style=CommunicationStyle
7     (api=APIDecl)?
8    '}'
9  ;
```

The `ServiceDecl` rule starts with the keyword `service` followed by the attribute `name` matched by the *textX* built-in rule `ID` that further follows the literal string match "{" (line 2). The body of the microservice declaration starts with the definition of two optional variables. First, we have a variable that keeps reference towards the configuration server. Its definition starts with the `config_server` keyword followed by the literal string match "=", after which comes the attribute `config_server` matched by the rule `FQN` (line 3). Second, we have a similarly defined variable that keeps reference towards the service registry (line 4). Then the optional variable assignment matched by the rule `Deployment` (line 5) follows. Next, we have the definition of communication style `CommunicationStyle` (line 6). In the end, another optional attribute `api` is matched by the rule `APIDecl` (line 7). The closing curly brace ends the microservice declaration.

Listing 2 shows how to define a simple `User` microservice in Silvera. `User` microservice is registered within `ServiceRegistry` (line 3), and it communicates with the rest of the system by using RPC (line 4). This microservice is deployed inside a container, and it listens to HTTP requests on the 8080 HTTP port. Since the `host` attribute is not defined in the `deployment` section, its default value will be applied—http://localhost, accessed on 1 June 2022. The API of this microservice consists of the `User` domain object and several publicly available methods. CRUD methods for the `User` domain object are generated automatically due to the `@crud` annotation. This annotation represents a shortcut, and the same effect can be achieved by using `@create`, `@read`, `@update`, and `@delete` annotations. In addition to the CRUD methods, we have three additional methods `listUsers`, `userExists`, and `userEmail`. All API methods for this microservice are exposed over REST. URL mapping for each of the methods will be auto-generated based on the microservice URL, method name, and HTTP method defined by the corresponding `@rest` annotation. For example, the method `listUsers` can be accessed with the following URL: http://localhost:8080/user/listusers, accessed on 1 June 2022. It is, however, possible to set custom URL mapping for the API method by using the `mapping` attribute of the annotation: `@rest(method=GET,mapping=<user_defined_mapping>)`. This attribute, however, is not currently available when using CRUD annotations.

Listing 2. An example that shows how to define a service.

```
1  service  User {
2
3     service_registry=ServiceRegistry
4     communication_style=rpc
5
6     deployment{
7        version="0.1"
8        port=8080
9        host=container
10    }
11
12    api{
13       @crud
14       typedef User[
15            @id str username
16            @required str password
17            @required @unique str email
18
19            int age   //optional
20       ]
21
22       @rest (method=GET)
23       list<User> listUsers ()
24
25       @rest(method=GET)
26       bool userExists (str username)
27
28       @rest (method=GET)
29       str userEmail(str username)
30    }
31 }
```

3.3.2. Service Registry Declaration

Listing 3 shows how to define a service registry named `ServiceRegistry`. This service registry is generated as a `Eureka` service registry (line 2) that will not register itself within another service registry (line 3). `ServiceRegistry` listens for requests at the 9091 HTTP port at http://registry.example.com, accessed on 1 June 2022 and is deployed inside a container (line 8). The current version of the registry is `0.0.1`.

Listing 3. An example that shows how to define a service registry.

```
1  service-registryServiceRegistry{
2     tool=eureka
3     client_mode=False
4     deployment  {
5          version="0.0.1"
6          port=9091
7          url="http://registry.example.com"
8          host=container
9     }
10 }
```

3.3.3. API Gateway Declaration

Listing 4 shows how to define an API gateway named `EntryGateway`. The `EntryGateway` provides a single entry point to the system, and the user only needs to remember the URL of the gateway. For each microservice behind the gateway, an URL mapping is provided. For example, to call the `listUsers` method from the `User` microservice, a user needs to use the following call: http://entry.example.com:9095/api/u/listusers, accessed on 1 June 2022. The `EntryGateway` is also registered within `ServiceRegistry`.

Listing 4. An example that shows how to define an API gateway.

```
1  api-gateway  EntryGateway  {
2
3     service_registry=ServiceRegistry
4
5     deployment  {
6          version="0.0.1"
7          port=9095
8          url="http://entry.example.com"
9     }
10
11     communication_style=rpc
12
13     gateway-for{
14          User as  /api/u
15     }
16  }
```

3.3.4. Declaration of Microservice Dependency

Listing 5 shows how to define a dependency between `Order` and `User` microservices. In this particular example, the `Order` microservice requires `userExists` and `userEmail` methods from the `User` microservice. For each requirement, a failure recovery strategy is defined (a `fallback_static` strategy for the `userExists` method and a `fail_silent` strategy for the `userEmail` method).

Listing 5. Defining dependency between Order and User microservices.

```
1  dependency  Order -> User  {
2     userExists[fallback_static]
3     userEmail[fail_silent]
4  }
```

3.3.5. Switching from RPC to Messaging Communication

So far, we have shown how to define microservices that use the RPC mechanism to communicate. In the text that follows, we will show how to change communication style from RPC to messaging.

First, the message pool and a message broker need to be defined. Listing 6 shows how to define the message pool with one message group —`UserMsgGroup`. This message group contains three message types: `UserAdded`, `UserUpdated`, and `UserDeleted`. Each message has two fields: `userId` and `userEmail`.

Listing 6. An excerpt of the grammar rule for microservice declaration.

```
msg-pool{
  group UserMsgGroup [
    msg UserAdded [
     str userId
     str userEmail
    ]
    ...
  ]
}
```

Listing 7 shows how to define a message broker named `Broker`. This message broker has three typed message channels: `EV_USER_ADDED_CHANNEL` channel for the `UserAdded` message, `EV_USER_UPDATED_CHANNEL` channel for the `UserUpdated` message, and `EV_USER_DELETED_CHANNEL` channel for the `UserDeleted` message. When instantiating a channel, the FQN of a message must be used.

Listing 7. An example that shows how to define a message broker.

```
msg-broker Broker  {

    channel EV_USER_ADDED_CHANNEL(UserMsgGroup.UserAdded)
    channel EV_USER_UPDATED_CHANNEL(UserMsgGroup.UserUpdated)
    channel EV_USER_DELETED_CHANNEL(UserMsgGroup.UserDeleted)
}
```

Listing 8 shows how the `User` microservice should be changed to use the messaging communication style. In the example, the User microservice publishes a message every time a user is added, updated, or deleted. Not only CRUD methods can produce or consume messages; regular API functions can use `@producer` and `@consumer` annotations that use the same syntax as shown in the example.

Listing 8. User microservices that uses messaging communication style.

```
serviceUser  {
  ...
  communication_style=messaging
  ...

  api {
      @create(UserMsgGroup.UserAdded -> Broker.EV_USER_ADDED_CHANNEL)
      @read
      @update(UserMsgGroup.UserUpdated ->
           Broker.EV_USER_UPDATED_CHANNEL)
      @delete(UserMsgGroup.UserDeleted ->
           Broker.EV_DELETED_DELETED_CHANNEL)
      typedef User [
         ...
       ]
       ...
   }
}
```

3.4. Compiler

Silvera compiler consists of two logically separated parts: *front-end* and *back-end*. The *front-end* further consists of modules for analysis and evaluation, whereas the *back-end* is comprised of a set of language-specific code generators.

3.4.1. Front-End

The compiler's *front-end* performs lexical analysis, parsing, semantic analysis, and translation to an intermediate representation. The parser is produced by *textX* [52] based on the Silvera grammar. The textX is an open-source tool for fast DSL development in Python that is IDE agnostic and provides a fast round-trip from grammar change to testing [52]. Since DSLs are susceptible to changes [48], we chose *textX* because it provides easy language evolution. The parser created by *textX* parses Silvera programs and creates a graph of Python objects (model) where each object is an instance of a corresponding class from the metamodel. This way, instead of an *Abstract Syntax Tree* (AST), textX returns a *Model* object (Section 3.2).

The front-end detects both syntax and semantic errors. Since Silvera is IDE-independent, errors are detected only during the compilation time. Syntax errors are detected early on, during parsing, by *textX*, which comes with extensive error reporting and debugging support [52].

Before the *Model* object is passed to the compiler's back-end, it is processed by the *communication resolving processor* (CRP) and the *architecture evaluation processor* (AEP). Since microservices created in Silvera can have multiple communication styles, the primary purpose of the CRP is to validate the Silvera model according to the corresponding communication style and enrich the model with communication-style-specific information needed by the compiler's back-end. Each communication style comes with a specific CRP.

The purpose of the AEP is to provide a metrics-based evaluation of the microservices-based system implemented in Silvera. Evaluation metrics applicable to microservice-based systems are defined by Bogner et al. [46]. Even though the AEP is an independent module, by utilizing it in the front-end, we are implementing the *AVOPT* pattern (see Section 3.1). This way, evaluation results can be used by the back-end to generate optimized applications.

For the evaluation, the AEP is using the following metrics: *Weighted Service Interface Count (WSIC)*, *Number of Versions per Service (NVS)*, *Services Interdependence in the System (SIY)*, *Absolute Importance of the Service (AIS)*, *Absolute Dependence of the Service (ADS)*, and *Absolute Criticality of the Service (ACS)*.

$WSIC(S)$ [53] is the number of exposed API functions of microservice S. Lower values for $WSIC$ are more favorable for the maintainability of a microservice. As absolute values for this metric are not conclusive on their own [46], the system-wide average $WSIC_{AVG}$ is calculated. By comparing values with the average, the largest microservices in the system can be identified and potentially split.

$NVS(S)$ [53] is the number of versions of microservice S currently used in the system. A large NVS_{AVG} value indicates high complexity and bears down on the maintainability [46].

SIY [54] is the number of microservice pairs that are bi-directionally dependent on each other. According to [54], interdependent pairs should be avoided as they attest to poor services' design. If such pairs exist, it can be a feasible solution to merge each of them into a single microservice [46].

$AIS(S)$ [54] is the number of consumer microservices that depend on the microservice S. AIS of every microservice is compared to a system-wide AIS_{AVG}, which can be used to identify very important microservices in the system.

$ADS(S)$ [54] is the number of microservices that microservice S depends on. Again, ADS_{AVG} is calculated and can be used for comparison.

$ACS(S)$ combines $AIS(S)$ and $ADS(S)$ to find the most critical and potentially problematic parts of the system. According to [54], the most critical microservices are those that are called from many different clients as well as those that invoke a lot of other microservices.

Evaluation is performed during the model compilation or on command (`silvera evaluate <path_to_model_dir>`). At the end of the evaluation, the AEP creates a detailed report file that contains calculated values for each microservice.

If needed, Silvera allows developers to register custom AEPs as plugins (see Section 3.4.3).

3.4.2. Back-End

After the front-end processes the *Model* object, the object is being passed to the compiler's back end as an input.

The back-end of the Silvera compiler iterates over each module in the model and passes declarations (*Decl* objects) to code generators. The number of code generators is not limited. The Silvera compiler offers a possibility to register custom code generators as plugins.

For every REST-based microservice, the back-end generates an OpenAPI document named `openapi.json`. OpenAPI files provide information about where to reach an API, which operations are available, what are the expected inputs and outputs, etc.

The built-in code generator. The built-in code generator uses template-based model-to-text transformations to produce the Java applications based on the Spring Boot (Spring Boot—https://spring.io/projects/spring-boot (accessed on 24 December 2021)) framework. The template-based code generation is a synthesis technique that produces code from high-level specifications called *templates* [55]. A template is an abstract representation of the textual output it describes. It has a static part, text fragments that appear in the output "as is", and a dynamic part embedded with splices of meta-code that encode the generation logic [55]. Templates lend themselves to iterative development as they can be easily derived from examples. Each declaration has a corresponding set of templates. The appropriate set of templates is chosen based on the declaration type and its target programming language. In the case of the built-in code generator, the target language is always Java 17. Once the appropriate set of templates is chosen, the code generator analyses the declaration and extracts relevant data. The data are subsequently used to fill the dynamic parts of the template.

The built-in code generator generates a Spring Boot application for every microservice present in the model. Most of the code is generated automatically; however, in some cases, developers must implement business logic manually. To ensure that the manually added code is preserved between successive code generations, the built-in code generator implements the *generation gap* pattern [56]. The implementation of this pattern ensures that manually written code can be added non-invasively using inheritance, where the manually added classes inherit the generated classes. A guide on adding manual changes to the generated code is part of Silvera's documentation (Introduce manual changes to the generated code—https://alensuljkanovic.github.io/silvera/compilation/#introduce-manual-changes-to-the-generated-code (accessed on 1 June 2022)).

We adhered to the best practices defined by Hofmann et al. [57], so each generated Spring Boot application has the following modules: *domain*, *controller*, *repository*, and *service*. Microservices that use messaging communication style contain two additional modules: *config* and *messages*.

The *domain* module contains classes that specify the application's domain model (business entities). These classes are derived from the type definitions (`typedefs`) located in the microservice's API. The *service* module contains classes that specify applications' business rules. These modules contain two sub-modules: *base* and *impl*. The *base* module contains a definition of the Java interface with methods defined in API, whereas the *impl* module contains a class that implements the base interface. The *impl* module is different from the rest of the generated modules because files in the *impl* module preserve manual changes in the code between successive code generations. In contrast, the rest of the generated files are always rewritten.

The *repository* module contains the implementation of the *Repository* pattern in the form of *MongoRepository* provided by *MongoDB*. Currently, the built-in code generator by

default supports only MongoDB. However, support for an arbitrary database can be added either by extending the existing code generator or by registering a new one.

All messages defined in the message pool are generated as classes in the *messages* module. Messages are sent through the network as JSON objects.

The *config* module contains classes used to define how the generated microservice application will communicate with a message broker. These classes are: (i) the *KafkaConfig* class that defines how the application is registered within the Kafka cluster, and (ii) the *MessageDeserializer* class that defines how messages received as JSON objects will be transformed into message objects defined in the *messages* module. *MessageDeserializer* is optional and will be generated only if the application consumes messages from the message broker.

The *controller* module contains a class that specifies the REST API of the generated microservice application. The class contains methods that belong to both the public and internal API of the microservice. Methods from the internal API are private and cannot be accessed from the outside.

In addition to the modules mentioned above, *pom.xml* and *bootstrap.properties* files are generated for each microservice. The file *bootstrap.properties* is used to setup Spring Boot applications, whereas Maven uses *pom.xml* files to manage dependencies.

API gateways and service registries are also generated as separate Spring Boot applications. The API gateway is generated as a *Zuul Proxy* (Netlix Zuul—https://github.com/Netflix/zuul (accessed on 24 December 2021)) server. *Zuul* is a gateway service developed by Netflix, and it provides dynamic routing, monitoring, resiliency, security, etc. Generated code for the API gateway is simple because it contains only one class with the main function and the *application.properties* file, which defines how the API gateway will perform request routing and whether it will contact the service registry to retrieve the URL of the corresponding microservice. The service registry is also a simple application, with one class with the main function and the *bootstrap.properties* file.

The built-in code generator produces a special `run` script and a *Docker* file (if the deployment host is set to the `container`) for each microservice, API gateway, or service registry. The `run` script utilizes *Maven* to produce and run the jar file.

3.4.3. Customization Support

Silvera allows users to register new AEP or code generators as plugins. Both are registered in the same way, so for brevity, we will only describe the registration process for the new code generator.

A custom code generator needs to be implemented in Python. Silvera uses the `pkg_resources` module from `setuptools` (Python's *setuptools* module—https://setuptools.readthedocs.io/en/latest/ (accessed on 24 December 2021)) and its concept of extension point to declaratively specify the registration of the new code generator. Extensions are defined within the project's `setup.py` module. All Python projects installed in the environment that declare the extension point will be discoverable dynamically.

The registration of a new code generator is performed in two steps. The first step is to create an instance of the `GeneratorDesc` class. An instance of the `GeneratorDesc` class contains information about the code generator's target language, description, and the reference towards the function that should be called to perform code generation. As shown in Listing 9, this function has three parameters: `Decl` object, a path to the directory where code will be generated, and a flag that shows whether the code generator is run in debug mode.

Listing 9. Implementation of a *GeneratorDesc* object and the prototype of the *generate* function.

```
1  from silvera.generator.gen_reg import GeneratorDesc
2
3  def generate(decl, output_dir, debug):
4      """Entry point function for code generator.
5
6      Args:
7          decl(Decl): can be declaration of service registry or config
8                      server.
9          output_dir(str):outputdirectory.
10         debug(bool): True if debug mode activated. False otherwise.
11      """
12      ...
13
14  python = GeneratorDesc(
15      language_name="python",
16      language_ver="3.7.4",
17      description="Python 3.7.4 codegenerator",
18      gen_func=generate
19 )
```

The second step is to make the code generator discoverable by Silvera. To do this, we must register the `GeneratorDesc` object in the `setup.py` entry point named `silvera_generators`, as shown in. Listing 10.

Listing 10. Making the new code generator discoverable by Silvera by using *silvera_generators* entry point.

```
1  from setuptools import setup
2
3  setup(
4      ...
5      entry_points={
6        'silvera_generators':[
7        'python = pygen.generator:python'
8        ]
9     }
10 )
```

Silvera provides a command (`silvera list-generators`) that lists all registered generators in the current environment.

4. Eat and Drink Microservice Architecture Use Case

In this section, we present a case study in which we have used Silvera to specify the architecture of the *Eat and Drink* application. The Eat and Drink is a distributed application for ordering food and drinks. We used Silvera to model the architecture of the Eat and Drink and to produce runnable Java code. In Section 4.1, we describe each microservice in the system, whereas in Section 4.3 we present the structure of the generated Java code. We will also compare the number of lines of code needed to specify the architecture of Eat and Drink in Silvera to the number of lines of code needed to specify the same Eat and Drink application in Java programming language.

4.1. Eat and Drink Microservice Architecture

In this section, we describe each microservice present in the Eat and Drink and how they interact.

The Eat and Drink application consists of only five microservices and serves as a short demonstration of Silvera's capabilities. In the future, we plan to test Silvera on real industrial use cases (see Section 8). Microservices that compose the Eat and Drink application are shown in Table 10. APIs of all microservices are based on REST and can be used by external users. The architecture of the Eat and Drink application is illustrated in Figure 3.

To provide a single entry point for external users, we defined *EntryGateway* as an API gateway. All microservices are discoverable via the *ServiceRegistry*. In the following text, we describe the domain model and the API of each microservice in the system.

The *User* microservice has only one entity—*User* with the following attributes: *username*, *password*, and *email*. The API of the *User* microservice consists of CRUD methods and three additional methods: (i) *listUsers*—which retrieves data for all users in the system, (ii) *userExists*—which checks if the user with a given username exists in the system, and (iii) *userEmail*—which retrieves the user's email.

The *Meal* microservice has two entities: *Meal* and *Ingredient*. Each meal has its *name*, *price*, *description*, and a list of *ingredients*. For each ingredient, it is possible to specify a name and a measure. The API of the *Meal* microservice consists of CRUD methods defined for *Meal* entity and methods such as: (i) *listMeals*—which retrieves all meals from the database, (ii) *mealExists*—which for a given meal name checks if the meal exists in the database, (iii) *mealPrice*—which for a given meal name returns its price, and (iv) *getIngredientsForAMeal*—which for a given meal name returns the list of the ingredients needed to prepare the meal.

The *Storage* microservice has only one entity—*Ingredient* with attributes *name* and *measure*. The API of the *Storage* microservice consists of CRUD methods defined for the entity and two additional methods: (i) *list*—which retrieves the information about all ingredients from the database, and (ii) *takeIngredient*—which, if possible, takes the ingredient from the storage.

Table 10. Microservices that compose Eat and Drink application and their descriptions.

Microservice	Description
User	Provides user-related operations.
Meal	Provides means for adding meals to the menu.
Order	Provides means for creating orders.
Storage	Contains information about availability of ingredients used for preparing a meal.
EmailNotifier	Notifies users about the order status via email.

The *Order* microservice has two entities: *Order* and *OrderItem*. Every order has information about the user who created the order, a list of *OrderItem* entities, and a calculated price. *OrderItem* contains information about the meal that is ordered, such as the meal name and the amount. The API of the *Order* microservice contains CRUD methods for the *Order* entity and the *listOrders* method, which retrieves all orders from the database. Methods *createOrder*, *updateOrder*, and *deleteOrder* publish corresponding messages to the message broker after they are successfully completed. These messages are used by the *EmailNotifier* microservice to notify the user about the order status.

The *EmailNotifier* microservice has one *Notification* entity that contains an order ID and a user's email as attributes. The API of this microservice has *listNotifications* and *listHistoryForUser* methods. The first method retrieves all notifications from the database, whereas the second method retrieves notifications only for a selected user. In addition to these publicly available methods, this microservice also contains three internal API methods:

(i) *orderCreated*, (ii) *orderUpdated*, and (iii) *orderDeleted*. These methods are triggered after an order in the *Order* microservice is created, updated, or deleted.

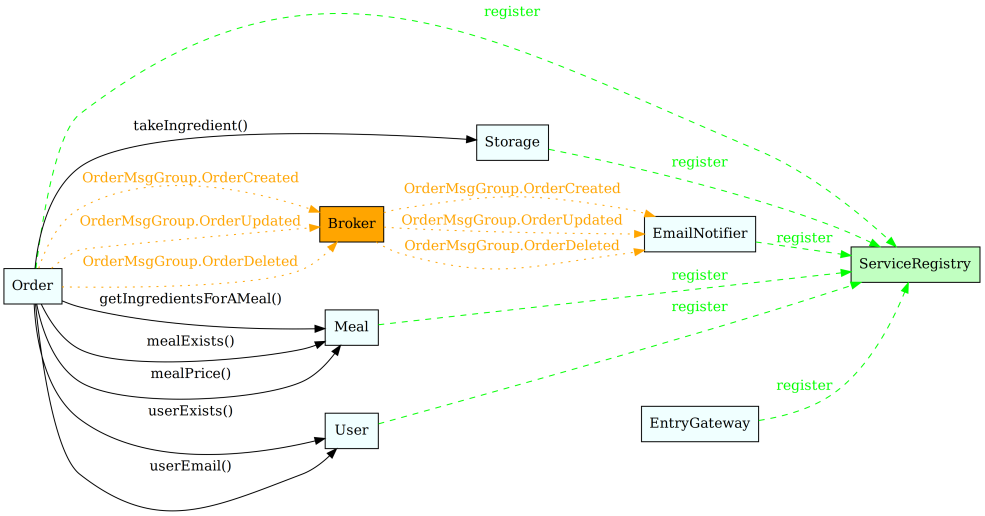

Figure 3. The architecture of the Eat and Drink application. Solid, black connections represent dependencies. The label on dependency connection shows which method is required by dependent. Green, dashed connections represent service registrations. Orange, dashed connections represent a message publishing, and labels on these connections represent messages that are being published.

4.2. Eat and Drink Architecture Evaluation

Silvera evaluated the Eat and Drink application during compilation. As shown by Table 11, there are a few outliers for metrics *WSIC*, *AIS*, and *ADS*.

WSIC values for microservices *User* and *Meal* are larger than the system average. Lower *WSIC* values are more favorable for the maintainability of a service. However, after we compared the APIs of all services in the system, we decided that the current implementation of *User* and *Meal* is good enough for this use case.

Table 11. The results of metrics-based evaluation of Eat and Drink application.

Metric	System Average	User	Meal	Storage	Order	EmailNotifier
WSIC	3.2	**4**	**5.5**	3	1	2.5
NVS	1	1	1	1	1	1
AIS	0.6	**1**	**1**	**1**	0	0
ADS	0.6	0	0	0	**3**	0
ACS	0	0	0	0	0	0

The microservice *Order* depends on three other microservices: *User*, *Meal*, and *Storage*. Based on the *ADS* value calculated for the *Order* microservice, we can see that this microservice could be problematic due to high coupling with its dependencies. However, this is expected since the *Order* microservice is a central part of the application where the main business logic is implemented. Since no other microservice depends on the *Order* microservice, its *AIS* and *ACS* values are both 0.

Microservices: *User*, *Meal*, *Storage*, and *EmailNotifier* have no dependencies towards other microservices, hence the zeros for their *ADS* and *ACS* values.

4.3. Eat and Drink Code Generation

As stated in Section 3.4, the Silvera compiler can have an arbitrary number of code generators, each specialized for a particular programming language or framework. In the case of *Eat and Drink*, we used the built-in code generator.

For each microservice in Eat and Drink, we compared the number of automatically generated lines of Java code and the number of manually added lines of Java code. First, we generated the code and counted the number of generated lines of code. We used *cloc* (cloc—https://github.com/AlDanial/cloc (accessed on 15 January 2021)) to calculate the number of lines of code for each microservice application. Then, we added the manual implementation where needed and ran *cloc* again to calculate the differences. As shown by Table 12, the majority of the code was generated automatically, whereas a small amount of code needs to be manually written.

Table 12. The comparison of the number of automatically generated lines of code and manually written lines of code for the Eat and Drink application.

Service	Silvera	Java		Maven		Sum			% Generated
		Generated LOC	Manually Added LOC	Generated LOC	Manually Added LOC	Generated LOC	Manually Added LOC	Total	
User	19	159	8	77	0	236	8	244	96.72
Meal	26	204	10	77	0	154	10	164	93.9
Storage	16	142	9	77	0	154	9	163	94.48
Order	37	523	27	81	0	162	27	189	85.71
EmailNotifier	24	379	59	81	4	162	63	225	72
ServiceRegistry	10	11	0	62	0	124	0	124	100
EntryGateway	23	14	0	60	0	120	0	120	100

Kelly and Tolvanen [58] have shown that DSLs lead to better quality applications because of two main reasons. First, DSLs can include correctness rules of the domain that ensure that the user cannot create illegal specifications. Elimination of bugs in the beginning is far better than finding and correcting them later [58]. Second, code generators automatically convert DSL specifications to a lower abstraction level (normally code), and the generated result does not need to be edited afterward. A study by Kieburtz et al. [59] compared the reliability of the software built manually and by using the DSL approach. The study compared the number of failed tests for the manual approach and the DSL approach and showed that the DSL approach yielded significantly fewer errors.

Based on these studies and the results from Table 12, we can conclude that Silvera, while accelerating the development of microservice-based distributed systems, also leads to better quality applications. The improved quality comes as a result of: (a) correctness rules built into Silvera that ensure that the user cannot create illegal specifications, and (b) the fact that most of the code is generated automatically and does not need to be edited afterward.

To check Silvera's performance, we compiled the Eat and Drink compilation 10 times in a row and collected the results with the built-in Linux shell *time* command. Results showed that it takes approximately 0.5 s to compile the application (The test was executed on a laptop with Intel Core™ i7-6700HQ CPU and 8 Gb of RAM).

5. Evaluation of Silvera

In addition to the Eat and Drink application presented in Section 4, we developed multiple applications to test Silvera. However, to achieve an objective assessment of the language, we needed feedback from users that were not involved in the development of Silvera. Therefore, we conducted a survey based on the FQAD framework [13] to understand whether we achieved our goals. FQAD is based on the ISO/IEC 25010:2011 standard, and it defines a set of quality characteristics that should be considered when creating a DSL. Many stakeholders can be involved in the assessment of the DSL. Each stakeholder forms a perspective of what characteristics the DSL should have [13]. Stakeholders can choose between

the following characteristics: functional suitability, usability, reliability, maintainability, productivity, extendability, compatibility, expressiveness, reusability, and integrability. The evaluation is based on the methodology presented by Wohlin et al. [60], whereas reporting is based on work presented by Jedlitschka et al. [61].

5.1. Scoping

The scope of the experiment was set by defining its goals [60]. Here we follow the GQM template for goal definition, originally presented by Basili and Rombach [62].

The goal of the study was to analyze Silvera for the purpose of evaluation with respect to the following DSL quality characteristics: functional suitability, usability, reliability, productivity, extendability, and expressivenes. The perspective is from the researcher's point of view. The study was run using students and software developers with experience in MSA or software architectures in general.

5.2. Hypotheses

Before we shared the survey with participants, we defined the following hypotheses.

Hypothesis 1. ***Silvera is not appropriate for the specification of MSA***. *Alternative Hypothesis $H1_1$: Silvera is appropriate for the specification of MSA.*

Hypothesis 2. ***Silvera language elements are not understandable***. *Alternative Hypothesis $H1_2$: Silvera language elements are understandable.*

Hypothesis 3. ***The concepts and notation of the Silvera language are not learnable and rememberable***. *Alternative Hypothesis $H1_3$: The concepts and notation of the Silvera language are learnable and rememberable.*

Hypothesis 4. ***Silvera is not appropriate for users' needs when developing MSA***. *Alternative Hypothesis $H1_4$: Silvera is appropriate for users' needs when developing MSA.*

Hypothesis 5. ***Silvera does not protect users against making errors***. *Alternative Hypothesis $H1_5$: Silvera protects users against making errors.*

Hypothesis 6. ***Silvera does not shorten the time needed to develop MSAs***. *Alternative Hypothesis $H1_6$: Silvera shortens the time needed to develop MSAs.*

Hypothesis 7. ***Silvera does not reduce the number of human resources used to develop MSAs***. *Alternative Hypothesis $H1_7$: Silvera reduces the number of human resources used to develop MSA.*

Hypothesis 8. ***Silvera does not provide one and only one good way to express every concept of interest.*** *Alternative Hypothesis $H1_8$: Silvera provides one and only one good way to express every concept of interest.*

Hypothesis 9. ***Each construct in Silvera is not used to represent exactly one distinct concept in the domain***. *Alternative Hypothesis $H1_9$: All constructs in Silvera represent exactly one distinct concept in the domain.*

Hypothesis 10. ***Silvera contains conflicting elements***. *Alternative Hypothesis $H1_{10}$: Silvera does not contain conflicting elements.*

Hypothesis 1 tests the *appropriateness* of Silvera. *Appropriateness* is a sub-characteristic of the *functional suitability*. Functional suitability refers to the degree to which a DSL is fully developed [13]. DSL should contain all necessary functionality from the domain and should not contain nor include functionality that is not in the domain [13].

Hypotheses 2–4 test the *comprehensibility*, *learnability*, and *likeability/user perception*, respectively. In FQAD, these characteristics are sub-characteristics of *usability*, which is defined as "the degree to which a DSL can be used by specified users to achieve specified goals" [13].

Hypothesis 5 tests the *reliability* of Silvera. Kahraman and Bilgen [13] define reliability of a DSL as "the property of a language that aids producing reliable programs (model checking ability/preventing unexpected relations)".

Hypotheses 6 and 7 test the *productivity*. Productivity is a characteristic related to the amount of resources expended by the user to achieve specified goals [13].

Hypotheses 8–10 test the *uniqueness*, *orthogonality*, and *conflicting elements*, respectively. In FQAD, these characteristics are sub-characteristics of *expressiveness*, which is defined as "the degree to which a problem solving strategy can be mapped into a program naturally" [13].

5.3. Selection of Participants

The target population of the study was software developers with experience in designing microservices. We also included developers that had significant experience in software architectures in general, but with lesser experience with microservices.

Participants were selected based on their availability and willingness to participate in the study. We did not offer any incentives for participation. All participants were identified through authors' professional networks. Participants were given 3 days to complete the study. Participants completed the study at their homes, so that they could work at their own pace. Participants were provided with video tutorials (Silvera tutorials—https://www.youtube.com/playlist?list=PLY6VwunTqUKdCudkeFkWXuUcRRcZrpSkL (accessed on 4 April 2022)), documentation (Silvera documentation—https://alensuljkanovic.github.io/silvera/ (accessed on 1 June 2022)), examples (Silvera examples—https://github.com/alensuljkanovic/silvera-demo (accessed on 4 April 2022)), and authors' email where they could ask questions. After studying the provided materials, participants were asked to implement a small microservice-based application both using the Silvera and manually (see Section 5.4).

This survey was not anonymous for two reasons. First, we wanted to make sure that each participant could only participate in the survey once. Second, we needed to be able to contact participants and ask for further explanations if they were needed. All participants were informed about the terms of participating in the survey, and we guaranteed that personal data would not be shared with third parties.

5.4. The Task

The task given to the participants is defined as follows:

> *Implement a distributed system based on microservices for publishing scientific papers. The system contains the following microservices:* `User`, `SciPaper`, *and* `Library`.
>
> *The* `User` *microservice allows users to register and log in. When registering, a user provides a username (ID), password (mandatory), first name (mandatory), last name (mandatory), and email (optional). To log in, the user must provide a username and password.*
>
> *The* `SciPaper` *microservice allows logged-in users to write new scientific papers. Each paper has an author, a title, and an arbitrary number of sections. Each section has its name and content. This microservice has a special method,* `publish`*, which, for a given paper ID, publishes a* `PUBLISH_PAPER` *message to a message broker. The message contains the paper's ID, paper's title, and author.*
>
> *The* `Library` *microservice has a method that listens to the* `PUBLISH_PAPER` *message and keeps data provided via a message in the database. This microservice also has a public method that lists all data from the database.*
>
> *The user is not aware of the application's architecture and uses the API gateway as an entry point. All services are registered in the service registry.*

Even though the task is simple, it covers all the basic concepts of Silvera. To successfully solve the task, participants needed to learn how to define a microservice and its API, an API gateway, and a service registry. Then, they needed to determine how to register a microservice within the service registry and how to make the microservice available in the API gateway. To successfully implement the business logic, participants needed to use both messaging and RCP communication styles. For the messaging communication style, they needed to define messages and a message pool. To make the RPC work, participants needed to define dependencies between the appropriate microservices.

The task needed to be implemented both in Silvera and manually. In the case of manual implementation, participants had the freedom to choose a programming language, framework, and tooling in which they implemented the task. Participants were asked to record the time needed to complete each implementation.

5.5. The Questionnaire

Once the example was implemented, participants were asked to answer the online questionnaire, which consisted of 13 questions with answers presented in the form of a five-level Likert scale. Participants could select only one answer. The questions were split into the following groups: *Experience*, *Functional suitability*, *Usability*, *Reliability*, *Productivity*, *Extendability*, and *Expressiveness*. The questions are presented in Table 13.

Table 13. The survey questions. All questions were closed type, and only one answer could be provided.

No.	Question Group	Question
Q1	Experience	How would you describe your previous experience in designing a software architecture?
Q2	Experience	How would you describe your previous experience in designing MSAs?
Q3	Experience	How would you describe you previous experience with CASE tools?
Q4	Functional suitability	Silvera is appropriate for microservice architecture specification needs.
Q5	Usability	Silvera language elements are understandable.
Q6	Usability	The concepts and notation of Silvera language are learnable and rememberable.
Q7	Usability	Silvera meets the user's needs for MSA specification.
Q8	Reliability	Silvera protects users against making errors via built-in validation.
Q9	Productivity	Silvera shortens the time needed to develop MSAs.
Q10	Productivity	Silvera reduces the amount of human resources used to develop MSAs.
Q11	Expressiveness	Silvera provides one and only one good way to express every concept of interest.
Q12	Expressiveness	Each construct in Silvera is used to represent exactly one distinct concept in the domain.
Q13	Expressiveness	Silvera does not contain conflicting elements.

For questions from the *Experience* group, users were able to select answers ranging from one, *Inexperienced*, to five. *Experienced*. For the rest of the questions, answers ranged from 1, *Strongly disagree*, to 5, *Strongly agree*. At the end of the questionnaire, we added a special field where participants could leave their comments. The data collected from the Likert scale in the study are quantitative.

5.6. Order and Completeness Rules

Before the survey started, we defined the following rules to prepare the environment.

Task implementation order. Each participant implemented the task (Section 5.4) twice. One group of participants was instructed to use Silvera first, whereas the other group was instructed to first implement the task manually in the programming language of their choice. We used randomized design to assign participants to one of these groups. This way, we avoided the effect of participants performing better in the second round of task implementation because of their familiarity with the problem. The participants submitted implementations after every round.

Task completeness. To make sure that the study results were valid, we checked if submitted task implementations were functional. We created a set of automated tests that were performed for every submitted implementation. These tests were used to calculate the task completeness and were not shared with the participants.

5.7. Analysis and Results

In total, 50 people were invited to participate in the survey, of which 24 accepted the invitation. However, only 18 participants completed the survey. Of the participants, 4 were developers with at least 3 years of working experience, and 14 were students in their last year of bachelor studies. As shown in Table 14, the experience of participants varied, with ∼33% of participants declaring themselves as *relatively experienced* or *experienced* in designing software architectures, ∼17% declaring themselves as *relatively experienced* or *experienced* in designed MSA, whereas ∼5.5% of participants declaring themselves as *relatively experienced* or *experienced* with CASE tools. Since we wanted to make sure that Silvera can be easily used by developers inexperienced with MSA, the participants met the study requirements.

Table 14. Percentages of responses (n = 18) for *Experience* group of questions.

Question	Inexperienced (%)	Relatively Inexperienced (%)	Medium Experienced (%)	Relatively Experienced (%)	Experienced (%)
Q1: How would you describe your previous experience in designing a software architecture?	5.55	5.55	55.55	27.77	5.55
Q2: How would you describe your previous experience in designing a microservice software architecture?	33.33	33.33	16.67	11.11	5.55
Q3: How would you describe your previous experience with CASE tools?	50.00	38.89	5.55	5.55	0

5.7.1. Performance Results

In Table 15, we show the performance results with regard to effectiveness and efficiency. All study participants implemented the task (Section 5.4) both with Silvera and manually (see Table 15). Participants needed assistance 10 times, resulting in a mean of *0.98*, as depicted in Table 15. All assistance was performed verbally (online), during which the participants were instructed to pause the clock.

The mean task completion time when using Silvera was just below 3.5 h. On average, participants implemented the task ∼124% faster when using Silvera than when implementing the same task manually. We used R language to perform a paired samples *t*-test to compare the means of two samples of implementation durations. We tested if Silvera indeed accelerates the implementation of MSAs, and we obtained a *p*-value (p-value = 5.539×10^{-5}) that was less than the significance level ($\alpha = 0.01$). The test statistic and effect size values were $V = -5.185483$ and *0.91*, respectively. Because the participants'

expertise in MSAs varied to a great extent, high standard deviation values for the task completion times were expected.

The completeness was measured by running the set of automated tests. As depicted in Table 15, both approaches yielded similar results, with Silvera having an average task completion rate of ∼95% compared to the ∼91% of the manual approach. While the maximum task completion rate was the same for both approaches, the minimum completion rate was slightly higher for the manual approach. Values for the minimum completion rate were lower than expected because some participants did not name their functions as stated in the task, which caused the failure of some of the automated tests. The *median* value was the same for both approaches. However, the *mode* value (the most commonly occurring value) showed that the task completion rate was 100% for the majority of participants when using Silvera. Indeed, 50% of the participants had a task completion of 100% when using Silvera, as opposed to 33.33% with the manual approach. We did not calculate the mode value for columns related to time because all time durations reported by users were unique.

Table 15. Performance results (effectiveness and efficiency).

	Mean	Median	Mode	Std. Dev.	Min	Max
Task completion rate—Silvera	94.84%	89.26%	100%	8.06%	71.42%	100%
Task completion rate—manual	91.26%	89.26%	85.71%	7.96%	78.57%	100%
Number of assistances	0.55	2	0	0.98	0	3
Time to implement the task—Silvera (hh:mm:ss)	03:21:56	03:08:12	-	01:16:07	01:13:24	06:45:23
Time to implement the task—manual (hh:mm:ss)	07:30:31	06:00:05	-	03:48:31	02:55:25	17:23:20

As hown in Table 16, the majority of participants agreed that Silvera is functionally suitable for the specification of distributed systems based on microservices. However, participants also pointed out the following problems:

- *Syntax error*. One participant reported that the generated code could not be compiled due to a syntax error. This was caused by faulty template implementation and was fixed immediately.
- *Code formatting*. A few participants complained about the formatting of the generated code. In some cases, the spacing between lines of code was too big. We addressed this problem, and we will pay special attention to the code formatting in the future.
- *Underutilized Spring Boot frame*. Some participants noted that Spring Boot comes with built-in features that could simplify some parts of the generated code. For example, we could use *Feign client* to implement the communication between microservices.
- *Notation consistency*. One participant commented that he was confused with the mixed usage of brackets and braces. We plan to take this into account in future versions of the tool.
- *Documentation*. One participant commented that video tutorials and documentation are not up-to-date. Video tutorials were recorded for the previous version of Silvera, but we did, however, describe all the backward-incompatible changes both in videos and the documentation. Still, we will address this issue in the future.

Table 16. Percentages of responses (n = 18) for *Functional suitability*, *Usability*, *Reliability*, *Productivity*, *Extendability* and *Expressiveness* groups of questions.

Question	Strongly Disagree (%)	Disagree (%)	Neutral (%)	Agree (%)	Strongly Agree (%)
Q4: Silvera is appropriate for MSA specification needs.	0	0	5.56	72.22	22.22
Q5: Silvera language elements are understandable.	0	0	0	27.78	72.22

Table 16. *Cont.*

Question	Strongly Disagree (%)	Disagree (%)	Neutral (%)	Agree (%)	Strongly Agree (%)
Q6: The concepts and notation of Silvera language are learnable and rememberable.	0	0	0	33.33	66.67
Q7: Silvera meets the user's needs for MSA specification.	0	0	11.11	55.56	33.33
Q8: Silvera protects users against making errors via built-in validation.	0	0	11.11	61.11	27.78
Q9: Silvera shortens the time needed to develop a microservice software architecture.	0	0	5.56	33.33	61.11
Q10: Silvera reduces the amount of human resources used to develop the microservice software architectures.	0	5.56	11.11	66.67	16.67
Q11: Silvera provides one and only one good way to express every concept of interest.	0	11.11	22.22	55.56	11.11
Q12: Each construct in Silvera is used to represent exactly one distinct concept in the domain.	0	0	11.11	77.78	11.11
Q13: Silvera does not contain conflicting elements.	0	0	5.56	72.22	22.22

5.7.2. Descriptive Statistics

We used the R language [63] to calculate the central tendency and dispersion of the responses. The *median* value (50%-percentile) for each question is either 4 or 5, as shown in Table 17. The *mode* value is also either 4 or 5, depending on a question. Median and mode values imply that the data is symmetrically distributed [60]. Low values for *Inter-Quartile Range* (IQR) indicate that there is a consensus among participants that Silvera satisfies the following DSL quality characteristics: functional suitability, usability, reliability, productivity, extendability, and expressiveness.

Table 17. Analysis of central tendency and dispersion of the responses (n = 18). Values for *median* and *mode* show the central tendency, while *Inter-Quartile Range* (IQR) shows the dispersion of the responses.

Question	Quantiles					Mode	IQR
	0(%)	25(%)	50(%)	75(%)	100(%)		
Q4: Silvera is appropriate for MSA specification needs.	3	4	4	4	5	4	0
Q5: Silvera language elements are understandable.	4	4.25	5	5	5	5	0.75
Q6: The concepts and notation of Silvera language are learnable and rememberable.	4	4	5	5	5	5	1
Q7: Silvera is appropriate for needs of MSA specification.	3	4	4	5	5	4	1
Q8: Silvera protects users against making errors.	3	4	4	4.75	5	4	0.75
Q9: Silvera shortens the time needed to develop a microservice software architectures.	3	4	5	5	5	5	1
Q10: Silvera reduces the amount of human resource used to develop the microservice software architectures.	2	4	4	4	5	4	0

Table 17. *Cont.*

Question	Quantiles					Mode	IQR
	0(%)	25(%)	50(%)	75(%)	100(%)		
Q11: Silvera provides one and only one good way to express every concept of interest.	2	3	4	5	5	4	1
Q12: Each construct in Silvera is used to represent exactly one distinct concept in the domain.	3	4	4	4	5	4	0
Q13: Silvera does not contain conflicting elements.	3	3	4	4	5	4	0

5.7.3. Hypothesis Testing

We used R language to perform a one-sample Wilcoxon signed-rank test [64] on our hypotheses. The one-sample Wilcoxon test is a rank-based test that calculates the difference between the observed and default values. We chose this test because the Likert scale data is ordinal and cannot be normally distributed. For each question in the questionnaire, we tested if the test result was greater than the default value of 3. In order to control the study-wide error rate, we used Bonferroni correction. For the Bonferroni correction, we multiplied the raw p-value by n to compute the corrected p-value, where n is the number of hypothesis tests in the study. In Table 18, we present the results of the test.

According to the data presented in Table 18, the p-value for each question is less than the significance level ($\alpha = 0.05$). Based on the results, we can reject the null Hypotheses (1–10) and accept the alternative Hypotheses ($H1_0$–$H1_{10}$). Thus, we can conclude that Silvera satisfies the following DSL quality characteristics: functional suitability, usability, reliability, productivity, extendability, and expressiveness.

Table 18. The corresponding test statistic, p-value and effect size for every question from groups *Functional suitability*, *Usability*, *Reliability*, *Productivity*, *Extendability* and *Expressiveness*.

Question	Test Statistic	p-Value	Effect Size
Q4: Silvera is appropriate fo MSA specification needs.	V = 153	0.0008	0.98
Q5: Silvera language elements are understandable.	V = 171	0.0006	1.00
Q6: The concepts and notation of Silvera language are learnable and rememberable.	V = 171	0.0006	0.99
Q7: Silvera is appropriate for needs of MSA specification.	V = 136	0.002	0.93
Q8: Silvera protects users against making errors.	V = 153	0.002	0.94
Q9: Silvera shortens the time needed to develop a microservice software architectures.	V = 129	0.001	0.96
Q10: Silvera reduces the amount of human resource used to develop the microservice software architectures.	V = 92	0.004	0.86
Q11: Silvera provides one and only one good way to express every concept of interest.	V = 136	0.045	0.68
Q12: Each construct in Silvera is used to represent exactly one distinct concept in the domain.	V = 153	0.0008	0.97
Q13: Silvera does not contain conflicting elements.	V = 120	0.0007	0.98

6. Threats to Validity

The aim of our study was to develop a language that will accelerate the development of microservices. Although the study conducted a thorough survey, there are some threats to validity that we will discuss in this section. We distinguish between threats to construct validity, internal validity, and external validity.

6.1. Construct Validity

Hypothesis guessing. Even though participants were not aware of the hypotheses, still, they could have answered the questionnaire in a way that would favor Silvera. Before the survey, we encouraged participants to give honest answers because that is the only way for Silvera to improve, and we feel that this threat was mainly mitigated.

Evaluation apprehension. A form of human tendency is to try to look better when being evaluated which confounds the outcome of the experiment [60]. The fact that the study was not anonymous could also affect the evaluation apprehension. To avoid this threat to validity we informed participants that were were not evaluating them; rather, we were evaluating Silvera. In addition, participants were not aware of other participants nor their results during the survey (each participant worked at their own home and at their own pace) to avoid any form of competition between them. In the future, we plan to enforce anonymity in contributions to further reduce evaluation apprehensions.

6.2. Internal Validity

Maturation. To answer the questionnaire, participants first needed to familiarize themselves with Silvera either by watching the tutorials or by reading the documentation and going through examples. After that, they needed to implement a simple task. All these steps required a significant portion of their time and could lead to boredom and disinterest. To mitigate this, we designed the task to be as simple as possible but also structured the documentation so that participants could quickly setup the environment and solve the task (see Section 5.4).

Instrumentation. In Section 4.3 we showed that most of the code for the presented use-case is generated automatically by Silvera and does not need to be edited afterwards. Nevertheless, the ratio between the amount of generated code and the manually written code depends on both the developer's coding style and level of expertise. In addition, results may vary depending on the tool used to count the number of lines in the code. Furthermore, even though the presented use-case covered all concepts of Silvera, the number of manual changes may be different for a different microservice-based system.

6.3. External Validity

Selection. While conducting the survey, we managed to gather only 18 participants to evaluate Silvera. That might be due to the significant time needed for participants to familiarize themselves with Silvera to complete the questionnaire. In addition, we aimed to select participants with previous experience in working with microservices, but participants' experience varied. Therefore, the survey may yield different results in the case of a larger number of participants, or in case of a larger number of more experienced participants.

Setting. The complexity of the task performed by participants during the survey may also affect survey results. The task was designed to be simple but still to cover all basic concepts of Silvera (see Section 5.4). That allowed all participants to complete the task in a reasonable time. However, a simple task could affect the results in many ways. For example, whether the DSL is appropriate for the domain ($H0_1$) and DSL's usability ($H0_2 - H0_4$) are better tested on more complex tasks. Additionally, the number of errors in simple tasks is less than in more complex tasks due to fewer LOC. Thus, how Silvera protects users from making mistakes ($H0_5$), productivity gains, ($H0_6$) and resource allocation ($H0_7$) could be better perceived in the case of a more complex task. Lack of IDE support could also affect the results of the study. Participants could use the arbitrary text editor to create Silvera programs, but since Silvera is not integrated with any of them, errors can be discovered

only during the compilation. To mitigate this, we paid special attention to error reporting and performance (see Section 4.3) when developing Silvera. This enabled participants to quickly discover and fix errors. However, we plan to add IDE support in the future to enhance the user experience (see Section 8).

Due to presented threats to external validity, we cannot claim that the results of the study can be generalized to other participant populations or settings. To mitigate this, we plan to perform more detailed studies in future (see Section 8).

7. Study Limitations

In this section, we discuss the limitations of the current approach and implementation. In addition, we give some ideas for future work.

7.1. Decentralized Development

Silvera performs at its best if the whole application is described in the model. However, Silvera is not necessarily a centralized solution. Some teams may opt for Silvera during the development of MSA, whereas others may choose the manual approach. This approach has several limitations since the information about the system can be dispersed over several Silvera models or manually implemented microservices.

Lack of information about dependencies. Microservices defined outside of the Silvera model cannot be set as dependencies of microservices from the model. Because the AEP (see Section 3.4.1) relies on dependencies, the results of the architecture evaluation will be incomplete or incorrect. Silvera generates a code for communication between the microservices based on the dependencies. In this case, such code needs to be added manually.

A service registry is not part of the model. In this case, the service registry is inside another model or implemented manually. Microservices that are defined in the model where the service registry is not defined cannot be automatically registered within the service registry. Because of this, the code for the registration of the microservice needs to be added manually.

An API gateway is not part of the model. In this case, the API gateway configuration needs to be updated manually in case of API changes of a microservice developed in Silvera.

Incomplete API gateway definition. In this case, the API gateway and some microservices are in separate models, or some of them are implemented manually. Microservices that are defined in other models or manually are not visible to the API gateway. Because of this, the API gateway configuration needs to be updated manually in case of API changes.

Missing message definitions. If there are multiple Silvera models or some microservices are implemented manually, each part of the system may have only a small subset of the complete message set. Due to Silvera's validation, it is impossible to define the message producer or consumer if the message is not defined within the model's message pool. Because of this, each message pool needs to be updated with the missing messages. Since each model has its own message pool, one team is in charge of one model/message pool. Teams could communicate which messages are used through the documentation. While this approach solves the problem of missing messages, it may create another problem where the same message, due to redundancy, needs to be updated in multiple message pools.

While we have workarounds for the problems mentioned above, in the future, we plan to investigate how we can solve these problems systematically. For example, if the application is implemented via multiple Silvera models, we could add a mechanism for importing declarations (microservices, service registries, and others) from other models. This mechanism must work in cases where models are on remote machines and should not deteriorate the performance.

7.2. Lack of IDE Support

DSLs are most effective when supported by specialized tooling including, but not limited to, an integrated development environment (IDE) with editors tailored for the language.

Silvera currently lacks an IDE support. In case of complex models, this can significantly affect the user experience. Adding features such as auto-complete, go to definition, or documentation on hover for a programming language takes significant effort. Tradi-

tionally, this work had to be repeated for each development tool, as each tool provides different APIs for implementing the same feature. To support multiple IDEs, we plan to add support for the Language Server Protocol (LSP) (The Language Server Protocol—https://microsoft.github.io/language-server-protocol/overviews/lsp/overview/ (accessed on 18 May 2022)). The LSP standardizes the way in which a *language server*, which provides language data, and development tools communicate. For all *textX*-based DSLs, the language server is provided through *text–LS* (textX–LS—https://github.com/textX/textX--LS (accessed on 18 May 2022)) open–source project, and LSP support only needs to be added on the client side (text editor or IDE).

7.3. Business Logic Not Part of the Model

Due to its pluggable compiler back-end, Silvera can support an arbitrary number of code generators. Each code generator can produce the code for a specific target language. This allows users to quickly switch the implementation of a microservice from one target language to another (e.g., from Java to Python). However, business logic in the case of specific API methods still needs to be implemented manually. Since this is performed on the code level, these changes are not visible on the model level.

There are two approaches that we can take to allow the definition of the business logic in the model. First, we could expand Silvera with *action language*—an imperative language used for action definitions. This language should support the usual constructs such as declarations, statements, and expressions. A conventional critique of DSLs is that they require the redesign of such things as statements and expressions, which is hard to get right and complete [32]. Second, we could provide an option to *embed* the code of the target language in the model. This would give complete access to the full expressivity of the target language but would also hamper portability between different target languages.

7.4. Additional Microservice Design Patterns

In Section 2.2, we descibed most MSA design patterns currently implemented in Silvera. However, the number of implemented design patterns can be increased. In this section, we will discuss the patterns that are arguably missing.

Security. Although there are many advantages of using MSA in developing distributed systems, security is still one of the biggest challenges [65]. However, several important security properties emerge as side effects of microservice design [66]: (i) microservices usually have a smaller code base, which leads to a smaller attack surface, (ii) malicious changes introduced by an attacker to a specific microservice instance are unlikely to persist past redeployment, if microservice is immutable, (iii) microservices only access the data and services they need, which limits the damage should an individual service be compromised, and (iv) heterogeneity of technologies protects microservices against low-level exploitation. Still, the following security practices are implemented in the industry [66]: (i) Mutual Transport Layer Security (MTLS) with a self-hosted Public Key Infrastructure (PKI) as a method to protect all internal service-to-service communication, and (ii) Security tokens for local authentication. Currently, Silvera generates a code where these practices need to be implemented manually. Users can mitigate this by overriding a built-in code generator or registering a new one; however, we plan to expand Silvera to generate a code with all existing security best practices.

Data management. Transactions that span multiple microservices need to be implemented manually. This comes as a direct consequence of the inability to express the business logic in Silvera models (see Section 7.3). Once the business logic becomes part of the model, we can implement the support for design patterns such as Saga [30] and Command Query Responsibility Segregation (CQRS) [27], and API composer [27]. Additionally, microservices are designed to be stateless. Stateless microservices facilitate autonomy and isolation. However, distributed applications usually must maintain a global state (as shown by [67]). How a DSL can be used to negotiate a global state and reach a consensus in MSA is shown in [68].

7.5. Expanding Silvera

A DSL does not attempt to address all types of computational problems, not even large classes of problems from a particular domain [32]. A DSL is very expressive for the problems that fall in the domain and cannot be used for other problems. The language may be inadequate even for the problems that are on the edge of the domain. This gray area typically puts pressure on the DSL to grow beyond its (original) domain [32].

It is challenging to anticipate the set of concepts needed over the lifetime of a programming language since these concepts are often dependent on the new platforms that programs must interact with.

Silvera is designed in a way that allows it to grow over time to accommodate the needs of the domain of MSA. New modeling concepts can be added by expanding the grammar and the compiler. However, this can also be a problem. By introducing a large number of new concepts, the language may become bloated which can significantly affect its learnability and maintainability. Still, adding support for a specific technology or framework on the compiler level is easier as code generators can be registered as plugins (see Section 3.4.3).

7.6. The Built-In Code Generator

Model-to-text transformations. The built-in code generator (see Section 3.4.2) uses template-based model-to-text transformations to produce the Java applications based on the Spring Boot framework. A significant drawback of template-based code generation is that syntax checks can only be performed after generating a complete output file [12]. In addition, the generator does not have access to the structure of the generated code. This makes it impossible to apply transformations, e.g., instrumentation, to the generated code [32]. We chose the template-based approach because it allowed us to have a working code generator quickly. However, this is not the only approach. Users can use whatever approach suits their needs to implement generators and register them as plugins (see Section 3.4.3).

A feature not implemented for a target platform. There is always a possibility that the built-in code generator does not cover some of the features available in the Spring Boot framework. In this case, users can expand the existing generator or implement a new code generator.

7.7. Architecture Evaluation Metrics

Silvera uses specifically designed service-based systems metrics [46] to evaluate the modeled architecture. These metrics can be applied to microservices also [46].

To the best of our knowledge, these metrics have not been empirically evaluated across multiple industrial use-cases to determine if they are complete and appropriate for MSA. $WSIC(S)$ and $NVS(S)$ were evaluated in only one use-case but in the context of SOA [53]. $AIS(S)$, $ADS(S)$, and $ACS(S)$ metrics roughly correspond to "class coupling" and "object coupling" metrics from the *Object-oriented programming*. There are several empirical studies where "class coupling" and "object coupling" metrics are evaluated [69].

Several other sets of metrics that were either designed for MSA or are applicable to the style [70–72] could be used during the evaluation. Additionally, experienced developers may disregard the evaluation results on purpose. For example, a microservice can depend on many other microservices because it contains the implementation of the main business logic. Developers may even invent their own set of metrics that are applied in their projects.

To enable usage of a different set of evaluation metrics, Silvera allows developers to register custom AEPs as plugins (see Section 3.4.3).

7.8. Unit Tests

Having unit tests is crucial when developing software. Writing or updating the unit tests can be tedious work if the microservice is frequently changed.

In the current implementation, Silvera is not generating unit tests. By generating unit tests, Silvera could further reduce the need for manual intervention.

8. Conclusions and Future Work

In this paper, we have presented *Silvera* —a domain-specific language for modeling distributed systems based on microservices. When creating Silvera, we had several goals in mind. The first goal was to provide a simple notation for writing microservice specifications that is easy to learn, use, and comprehend for both experienced and inexperienced developers. A simple but semantically rich notation would provide a faster way to develop MSA systems. The second goal was to support heterogeneous technology stacks. Our aim was to allow developers to use the best tool for the job and also encourage experimentation. Frequent changes in MSA systems often lead to obsolete documentation. Hence, our third goal was to provide automatic generation of documentation. Finally, our fourth goal was to provide a mechanism for users to evaluate the architecture of the created systems.

We have accomplished our goals. Specifications written in Silvera are transformed to Java code by the Silvera compiler. To support multiple programming languages as a target, the compiler allows users to register custom code generators written in Python. For every microservice, Silvera generates documentation in form of an OpenAPI document. In addition, Silvera comes with an architecture evaluation processor inside the compiler that provides a metrics-based evaluation of the microservices-based system implemented in Silvera. If needed, custom evaluation processors can be registered as plugins.

We provided an example of how Silvera can be used in Section 4. To obtain objective evaluation results, we conducted a survey based on the FQAD framework. After analyzing the survey results, we concluded that Silvera satisfies required DSL quality characteristics.

In the future, we plan to perform more detailed studies with a larger number of participants. More specifically, we want to know how Silvera performs in a team setting, where each team will be given a specific microservice to implement, and how metrics-based evaluations help with improving the system. In addition, we would like to test Silvera in systems with very large numbers of microservices.

We plan to expand Silvera in several ways. We plan to support security for security and authentication services. In case of messaging-based communication, we plan to support the collection of lost messages (Dead Letter Queue) and error messages (Error Queue). Additionally, we will increase the number of supported target languages by adding new code generators. In addition, we will look into the possibility of modeling a business logic directly in Silvera.

To enhance the user experience, we plan to implement IDE support with features such as auto-complete, go to definition, documentation on hover, and others.

Silvera is a free and open-source project hosted on GitHub, and it is provided under the terms of the MIT license.

Author Contributions: Conceptualization, A.S. and I.D.; methodology, A.S., V.I. and I.D.; software, A.S.; validation, A.S., B.M., V.I. and I.D.; investigation, A.S., B.M., V.I. and I.D.; writing—original draft preparation, A.S.; writing—review and editing, B.M., V.I. and I.D.; visualization, A.S.; supervision, B.M. and I.D.; All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Silvera source code, examples, documentation and video materials are available at https://alensuljkanovic.github.io/silvera/, accessed on 2 June 2022. Additional information is available from corresponding authors upon a reasonable request.

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

References

1. Fowler, M.; Lewis, J. Microservices. 2014. Available online: https://www.martinfowler.com/articles/microservices.html (accessed on 30 April 2017).
2. Di Francesco, P.; Malavolta, I.; Lago, P. Research on architecting microservices: Trends, focus, and potential for industrial adoption. In *Software Architecture (ICSA), Proceedings of the 2017 IEEE International Conference on Software Architecture (ICSA), Gothenburg, Sweden, 3–7 April 2017*; IEEE: Piscataway, NJ, USA, 2017; pp. 21–30.
3. Fritzsch, J.; Bogner, J.; Wagner, S.; Zimmermann, A. Microservices migration in industry: Intentions, strategies, and challenges. In Proceedings of the 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), Cleveland, OH, USA, 29 September–4 October 2019; IEEE: Piscataway, NJ, USA, 2019; pp. 481–490.
4. Baškarada, S.; Nguyen, V.; Koronios, A. Architecting microservices: Practical opportunities and challenges. *J. Comput. Inf. Syst.* **2018**, *60*, 1–9. [CrossRef]
5. Bogner, J.; Fritzsch, J.; Wagner, S.; Zimmermann, A. Microservices in industry: Insights into technologies, characteristics, and software quality. In Proceedings of the 2019 IEEE International Conference on Software Architecture Companion (ICSA-C), Hamburg, Germany, 25–26 March 2019; IEEE: Piscataway, NJ, USA, 2019; pp. 187–195.
6. Knoche, H.; Hasselbring, W. Drivers and barriers for microservice adoption-a survey among professionals in germany. *Enterp. Model. Inf. Syst. Archit. (EMISAJ)—Int. J. Concept. Model.* **2019**, *14*, 1–35.
7. Wang, Y.; Kadiyala, H.; Rubin, J. Promises and challenges of microservices: An exploratory study. *Empir. Softw. Eng.* **2021**, *26*, 1–44. [CrossRef]
8. Lenarduzzi, V.; Lomio, F.; Saarimäki, N.; Taibi, D. Does migrating a monolithic system to microservices decrease the technical debt? *J. Syst. Softw.* **2020**, *169*, 110710. [CrossRef]
9. Kleehaus, M.; Matthes, F. Challenges in Documenting Microservice-Based IT Landscape: A Survey from an Enterprise Architecture Management Perspective. In Proceedings of the 2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC), Paris, France, 28–31 October 2019; IEEE: Piscataway, NJ, USA, 2019; pp. 11–20.
10. Bushong, V.; Abdelfattah, A.S.; Maruf, A.A.; Das, D.; Lehman, A.; Jaroszewski, E.; Coffey, M.; Cerny, T.; Frajtak, K.; Tisnovsky, P.; et al. On Microservice Analysis and Architecture Evolution: A Systematic Mapping Study. *Appl. Sci.* **2021**, *11*, 7856. [CrossRef]
11. Waseem, M.; Liang, P.; Shahin, M. A systematic mapping study on microservices architecture in devops. *J. Syst. Softw.* **2020**, *170*, 110798. [CrossRef]
12. Voelter, M.; Benz, S.; Dietrich, C.; Engelmann, B.; Helander, M.; Kats, L.C.; Visser, E.; Wachsmuth, G. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*; CreateSpace Independent Publishing Platform 2013. Available online: dslbook.org (accessed on 20 June 2022).
13. Kahraman, G.; Bilgen, S. A framework for qualitative assessment of domain-specific languages. *Softw. Syst. Model.* **2015**, *14*, 1505–1526. [CrossRef]
14. ISO/IEC/IEEE. *ISO/IEC/IEEE Systems and Software Engineering—Architecture Description*; ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000); IEEE: Piscataway, NJ, USA, 2011; pp. 1–46. [CrossRef]
15. Górski, T. The 1+5 Architectural Views Model in Designing Blockchain and IT System Integration Solutions. *Symmetry* **2021**, *13*, 2000. [CrossRef]
16. Richards, M. *Microservices vs. Service-Oriented Architecture*; O'Reilly Media: Newton, MA, USA, 2015.
17. Dragoni, N.; Giallorenzo, S.; Lafuente, A.L.; Mazzara, M.; Montesi, F.; Mustafin, R.; Safina, L. Microservices: Yesterday, today, and tomorrow. *arXiv* **2016**, arXiv:1606.04036.
18. Autili, M.; Di Salle, A.; Gallo, F.; Pompilio, C.; Tivoli, M. CHOReVOLUTION: Service choreography in practice. *Sci. Comput. Program.* **2020**, *197*, 102498. [CrossRef]
19. Serhani, M.A.; El-Kassabi, H.T.; Shuaib, K.; Navaz, A.N.; Benatallah, B.; Beheshti, A. Self-adapting cloud services orchestration for fulfilling intensive sensory data-driven IoT workflows. *Future Gener. Comput. Syst.* **2020**, *108*, 583–597. [CrossRef]
20. Cerny, T.; Donahoo, M.J.; Trnka, M. Contextual understanding of microservice architecture: Current and future directions. *ACM SIGAPP Appl. Comput. Rev.* **2018**, *17*, 29–45. [CrossRef]
21. Li, J.; Zhong, Y.; Zhu, S.; Hao, Y. Energy-aware service composition in multi-Cloud. *J. King Saud-Univ.-Comput. Inf. Sci.* **2022**, *in press*. [CrossRef]
22. Gorski, T.; Woźniak, A.P. Optimization of business process execution in services architecture: a systematic literature review. *IEEE Access* **2021**, *9*, 111833–111852. [CrossRef]
23. Bucchiarone, A.; Dragoni, N.; Dustdar, S.; Larsen, S.T.; Mazzara, M. From monolithic to microservices: An experience report from the banking domain. *IEEE Softw.* **2018**, *35*, 50–55. [CrossRef]
24. Namiot, D.; Sneps-Sneppe, M. On micro-services architecture. *Int. J. Open Inf. Technol.* **2014**, *2*, 24–27.
25. Dragoni, N.; Lanese, I.; Larsen, S.T.; Mazzara, M.; Mustafin, R.; Safina, L. Microservices: How to make your application scale. *arXiv* **2017**, arXiv:1702.07149.
26. Gabbrielli, M.; Giallorenzo, S.; Guidi, C.; Mauro, J.; Montesi, F. Self-reconfiguring microservices. In *Theory and Practice of Formal Methods*; Springer: Berlin/Heidelberg, Germany, 2016; pp. 194–210.
27. Richardson, C. *Microservice Patterns*; Manning Publications: Shelter Island, NY, USA, 2017.
28. Karabey Aksakalli, I.; Çelik, T.; Can, A.; Tekinerdogan, B. Deployment and communication patterns in microservice architectures: A systematic literature review. *J. Syst. Softw.* **2021**, *180*, 111014. [CrossRef]

29. Houmani, Z.; Balouek-Thomert, D.; Caron, E.; Parashar, M. Enhancing microservices architectures using data-driven service discovery and QoS guarantees. In Proceedings of the 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID), Melbourne, VIC, Australia, 1–14 May 2020; IEEE: Piscataway, NJ, USA, 2020; pp. 290–299.
30. Newman, S. *Building Microservices*; O'Reilly Media: Newton, MA, USA, 2015.
31. Van Deursen, A.; Klint, P.; Visser, J. Domain-specific languages: An annotated bibliography. *ACM Sigplan Not.* **2000**, *35*, 26–36. [CrossRef]
32. Visser, E. WebDSL: A case study in domain-specific language engineering. In *International Summer School on Generative and Transformational Techniques in Software Engineering*; Springer: Berlin/Heidelberg, Germany, 2007; pp. 291–373.
33. Kosar, T.; Lu, Z.; Mernik, M.; Horvat, M.; Črepinšek, M. A Case Study on the Design and Implementation of a Platform for Hand Rehabilitation. *Appl. Sci.* **2021**, *11*, 389. [CrossRef]
34. Dejanović, I.; Dejanović, M.; Vidaković, J.; Nikolić, S. PyFlies: A Domain-Specific Language for Designing Experiments in Psychology. *Appl. Sci.* **2021**, *11*, 7823. [CrossRef]
35. Wile, D. Lessons learned from real DSL experiments. In Proceedings of the 36th Annual Hawaii International Conference on System Sciences, Big Island, HI, USA, 6–9 January 2003; IEEE: Piscataway, NJ, USA, 2003; p. 10.
36. Gray, J.; Karsai, G. An examination of DSLs for concisely representing model traversals and transformations. In Proceedings of the 36th Annual Hawaii International Conference on System Sciences, Big Island, HI, USA, 6–9 January 2003; IEEE: Piscataway, NJ, USA, 2003; p. 10.
37. Johanson, A.N.; Hasselbring, W. Effectiveness and efficiency of a domain-specific language for high-performance marine ecosystem simulation: A controlled experiment. *Empir. Softw. Eng.* **2017**, *22*, 2206–2236. [CrossRef]
38. Kosar, T.; Gaberc, S.; Carver, J.C.; Mernik, M. Program comprehension of domain-specific and general-purpose languages: Replication of a family of experiments using integrated development environments. *Empir. Softw. Eng.* **2018**, *23*, 2734–2763. [CrossRef]
39. Wizenty, P.; Sorgalla, J.; Rademacher, F.; Sachweh, S. MAGMA: Build management-based generation of microservice infrastructures. In Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings, Canterbury, UK, 11–15 September 2017; ACM: New York, NY, USA, 2017; pp. 61–65.
40. Sorgalla, J. Ajil: A graphical modeling language for the development of microservice architectures. In Proceedings of the Microservices 2017 Conference, Extended Abstracts, Odense, Denmark, 25–26 October 2017.
41. Perera, K.; Perera, I. A Rule-based System for Automated Generation of Serverless-Microservices Architecture. In Proceedings of the 2018 IEEE International Systems Engineering Symposium (ISSE), Rome, Italy, 1–3 October 2018; IEEE: Piscataway, NJ, USA, 2018; pp. 1–8.
42. Terzić, B.; Dimitrieski, V.; Kordić, S.; Milosavljević, G.; Luković, I. Development and evaluation of MicroBuilder: A Model-Driven tool for the specification of REST Microservice Software Architectures. *Enterp. Inf. Syst.* **2018**, 1–24. [CrossRef]
43. Sorgalla, J.; Wizenty, P.; Rademacher, F.; Sachweh, S.; Zündorf, A. Applying Model-Driven Engineering to Stimulate the Adoption of DevOps Processes in Small and Medium-Sized Development Organizations. *SN Comput. Sci.* **2021**, *2*, 1–25. [CrossRef]
44. Montesi, F.; Guidi, C.; Zavattaro, G. Service-Oriented Programming with Jolie. In *Web Services Foundations*; Springer: Berlin/Heidelberg, Germany, 2014; pp. 81–107.
45. Rademacher, F.; Sachweh, S.; Zündorf, A. Aspect-oriented modeling of technology heterogeneity in microservice architecture. In Proceedings of the 2019 IEEE International Conference on Software Architecture (ICSA), Hamburg, Germany, 25–29 March 2019; IEEE: Piscataway, NJ, USA, 2019; pp. 21–30.
46. Bogner, J.; Wagner, S.; Zimmermann, A. Automatically measuring the maintainability of service-and microservice-based systems: A literature review. In Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement, Gothenburg, Sweden, 25–27 October 2017; pp. 107–115.
47. Spinellis, D. Notable design patterns for domain-specific languages. *J. Syst. Softw.* **2001**, *56*, 91–99. [CrossRef]
48. Mernik, M.; Heering, J.; Sloane, A.M. When and how to develop domain-specific languages. *ACM Comput. Surv. (CSUR)* **2005**, *37*, 316–344. [CrossRef]
49. Jalali, S.; Wohlin, C. Systematic literature studies: Database searches vs. backward snowballing. In Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, Lund, Sweden, 20–21 September 2012; IEEE: Piscataway, NJ, USA, 2012; pp. 29–38.
50. Fowler, M. *Domain-Specific Languages*; Addison-Wesley Professional: Boston, MA, USA, 2010.
51. Kosar, T.; Martı, P.E.; Barrientos, P.A.; Mernik, M. A preliminary study on various implementation approaches of domain-specific language. *Inf. Softw. Technol.* **2008**, *50*, 390–405. [CrossRef]
52. Dejanović, I.; Vaderna, R.; Milosavljević, G.; Vuković, Ž. TextX: A Python tool for Domain-Specific Languages implementation. *Knowl.-Based Syst.* **2017**, *115*, 1–4. [CrossRef]
53. Hirzalla, M.; Cleland-Huang, J.; Arsanjani, A. A metrics suite for evaluating flexibility and complexity in service oriented architectures. In *Proceedings of the International Conference on Service-Oriented Computing*; Springer: Berlin/Heidelberg, Germany, 2008; pp. 41–52.
54. Rud, D.; Schmietendorf, A.; Dumke, R. Product metrics for service-oriented infrastructures. In *Proceedings of the 16th International Workshop on Software Measurement and DASMA Metrik Kongress*; IWSM/MetriKon: Potsdam, Germany, 2006; pp. 161–174.
55. Syriani, E.; Luhunu, L.; Sahraoui, H. Systematic mapping study of template-based code generation. *Comput. Lang. Syst. Struct.* **2018**, *52*, 43–62. [CrossRef]

56. Vlissides, J. *Pattern Hatching: Design Patterns Applied*; Addison-Wesley Longman Ltd.: Boston, MA, USA, 1998.
57. Hofmann, M.; Schnabel, E.; Stanley, K. *Microservices Best Practices for Java*; IBM Redbooks: Armonk, NY, USA, 2017.
58. Kelly, S.; Tolvanen, J.P. *Domain-Specific Modeling: Enabling Full Code Generation*; Wiley–IEEE Computer Society Pr.: Hoboken, NJ, USA, 2008.
59. Kieburtz, R.B.; McKinney, L.; Bell, J.M.; Hook, J.; Kotov, A.; Lewis, J.; Oliva, D.P.; Sheard, T.; Smith, I.; Walton, L. A software engineering experiment in software component generation. In Proceedings of the IEEE 18th International Conference on Software Engineering, Berlin, Germany, 25–30 March 1996; IEEE: Piscataway, NJ, USA, 1996; pp. 542–552.
60. Wohlin, C.; Runeson, P.; Höst, M.; Ohlsson, M.C.; Regnell, B.; Wesslén, A. *Experimentation in Software Engineering: An Introduction*; Kluwer Academic Publishers: Alphen aan den Rijn, The Netherlands, 2000.
61. Jedlitschka, A.; Ciolkowski, M.; Pfahl, D. Reporting Experiments in Software Engineering. In *Guide to Advanced Empirical Software Engineering*; Springer: Berlin/Heidelberg, Germany, 2008; pp. 201–228.
62. Basili, V.R.; Rombach, H.D. The TAME project: Towards improvement-oriented software environments. *IEEE Trans. Softw. Eng.* **1988**, *14*, 758–773. [CrossRef]
63. R Core Team *R: A Language and Environment for Statistical Computing*; R Core Team: Vienna, Austria, 2013.
64. Woolson, R. Wilcoxon signed-rank test. *Wiley Encyclopedia of Clinical Trials*; Wiley: Hoboken, NJ, USA, 2007; pp. 1–3.
65. Ghosh, A.; Mukherjee, A.; Misra, S. SEGA: Secured Edge Gateway Microservices Architecture for IIoT-based Machine Monitoring. *IEEE Trans. Ind. Inform.* **2021**, *18*, 1949–1956. [CrossRef]
66. Yarygina, T.; Bagge, A.H. Overcoming security challenges in microservice architectures. In Proceedings of the 2018 IEEE Symposium on Service-Oriented System Engineering (SOSE), Bamberg, Germany, 6–29 March 2018; IEEE: Piscataway, NJ, USA, 2018; pp. 11–20.
67. Belafia, R.; Jeanjean, P.; Barais, O.; Le Guernic, G.; Combemale, B. From Monolithic to Microservice Architecture: The Case of Extensible and Domain-Specific IDEs. In Proceedings of the 2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), Fukuoka, Japan, 10–15 October 2021; IEEE: Piscataway, NJ, USA, 2021; pp. 454–463.
68. El-Ghareeb, H.A. Neutrosophic-based domain-specific languages and rules engine to ensure data sovereignty and consensus achievement in microservices architecture. In *Optimization Theory Based on Neutrosophic and Plithogenic Sets*; Elsevier: Amsterdam, The Netherlands, 2020; pp. 21–43.
69. Aggarwal, K.; Singh, Y.; Kaur, A.; Malhotra, R. Empirical Study of Object-Oriented Metrics. *J. Object Technol.* **2006**, *5*, 149–173. [CrossRef]
70. Athanasopoulos, D.; Zarras, A.V.; Miskos, G.; Issarny, V.; Vassiliadis, P. Cohesion-driven decomposition of service interfaces without access to source code. *IEEE Trans. Serv. Comput.* **2014**, *8*, 550–562. [CrossRef]
71. Engel, T.; Langermeier, M.; Bauer, B.; Hofmann, A. Evaluation of microservice architectures: A metric and tool-based approach. In *International Conference on Advanced Information Systems Engineering*; Springer: Berlin/Heidelberg, Germany, 2018; pp. 74–89.
72. Haupt, F.; Leymann, F.; Scherer, A.; Vukojevic-Haupt, K. A framework for the structural analysis of REST APIs. In Proceedings of the 2017 IEEE International Conference on Software Architecture (ICSA), Gothenburg, Sweden, 3–7 April 2017; IEEE: Piscataway, NJ, USA, 2017; pp. 55–58.