Jurnal Ilmu Komputer dan Informasi (Journal of Computer Science and Information)
15/2 (2022), 105-118. DOI: http://dx.doi.org/10.21609/jiki.v15i2.1065

# Towards Erlang-based ABS Microservices Framework for Software Product Line Development

Adrika Novrialdi*, Daya Adianto*, Aulia Rosyida*,
Priambudi Lintang Bagaskara*, Ade Azurat*

* Faculty of Computer Science, Universitas Indonesia, Depok, Indonesia
*Email:{adrika.novrialdi01, aulia.rosyida, priambudi.lintang}@ui.ac.id*
*{dayaadianto, ade}@cs.ui.ac.id*

**Abstract**

Software Product Line Engineering (SPLE) is one of the approaches that can manage the variability in developing sets of products. However, there is a need for development tools such as programming language and toolchain in realising SPLE. One language that supports the SPLE process is Abstract Behavioral Specification (ABS). ABS Microservices is one research that utilises ABS to create a web framework that supports the SPLE process. This framework uses ABS to generate Java-based applications. However, there is a need for renewal to the ABS Microservices framework. Deprecation of the Java backend of the ABS opens a new exploration of another web framework that uses other ABS backend languages. We present the ABS Microservices web framework based on Erlang OTP. We choose Erlang because it promises more efficient resource usage and the Erlang backend is one of the ABS backends with the most available features. This research aims to create an entry point for ABS Microservices to support more language. We use a case study and apply the six quality factors of software product line implementation to evaluate our framework. This research shows that the Erlang variant of ABS Microservices has less resource usage than the Java variant. Hence, this promises more options to develop product lines using ABS Microservices.

**Keywords**: *Software product line engineering, Web engineering, Delta Oriented Programming, Microservices*

## 1. Introduction

Software development transforms, grows, and changes continuously, including market and industrial aspects. Many companies or organisations need to develop applications to automate their business processes. Each organisation develops the required features for the application one at a time. This causes applications with similarities to repeat the development when building similar features with several functions tailored to the needs of each organisation. This variation is referred to as software diversity that leads to an increase in the complexity and impact production time, cost, and maintenance effort because of the large risk of system failure [1]. Failing to plan the diversity of systems will complicate further development. Incompatible modelling and specification techniques can lead to unfit software structure, such as improper abstractions [2] and disintegrate functionalities. The developer might need an extra effort to refactor the system if there are requirements changes later, leading to more time and cost.

Instead of making a particular system for each product, the system development in multiple products can be achieved more efficiently through the Software Product Line Engineering (SPLE) paradigm. This approach aims to improve the productivity and quality of the products based on variability and commonality of features. First, the commonality of features is handled by building a common platform of overall products included in the product family. Second, the variability of features to get personalised and the end product is generated

by applying mass customisation [3]. Combinations between the variability affect many aspects in the development process, such as flexibility, standardisation, and saving development costs. The modularity can be achieved by the decomposition of the system in terms of development view, process view, code view, and component framework aspect [3]. It will speed up the maintenance process and allow flexible configuration.

Web application is one of the most popular types of software nowadays. Many organisations use this type of software as their primary business support. As a result, the research on various web application aspects has increased in the past years. From aesthetics research [4], quality and credibility framework to evaluate website [5], continuous personalization research [6], to enhancement of data security [7]. Recently, the web application has been leaning toward microservices architecture. Microservices provide some benefits [8], thus making many organisations also consider the use of this architecture. Microservices is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms [8]. This style changes the development paradigm to creating an application as a set of (small) services instead of a single unit [8]. In the microservices architecture, the impact of software evolution, e.g. requirements changes – the effort to adapt the system to the new requirements – can be mitigated, as each service is modular. Usually, microservices are independent of each other, allowing developers to freely choose and combine different technologies regarding, for instance, programming languages, databases or communication protocols [9]. Therefore, microservices are highly interoperable, enabling developers to integrate functionalities of different systems that are not implemented with the same technologies [10].

There are some challenges in implementing the SPLE process for web application development related to the integration of the SPLE process into traditional web engineering. SPLE process' objective is to create a platform to create multiple products as a product line opposite to the traditional software development, which creates just a specific application. In order to adapt the SPLE process in web engineering, there is a need to manage the requirements and artefacts of multiple applications instead of the single application in traditional web development. However, there are also many types of artefacts in a web application structure, and these artefacts are not only the ones directly related to the features of that web application but also the artefacts that serve as the supporting files such as web server and configuration files. The management of this kind of application artefact is another challenge of the implementation of SPLE in web applications.

SPLE research also starts to consider the study in web engineering and microservices. Aziz et al. [11] and Naily et al. [12] use Abstract Behaviour Specification (ABS) [13] to build an SPLE-enabled web framework, a web framework that supports the SPLE process. ABS [13] is a modelling languages that support the SPLE process. To create a product using ABS, a model is need to be defined. Then by applying SPLE process a product is generated. Aziz et al., and Naily et al., utilised the ABS language mechanism to create a web application by building a generator to generate a web application based on the model. The previous research from [11] and [12] use the Java backend of the ABS language, which generate a Java-based web framework. However, a specific programming language has its strength and weakness. For example, a language may suit a particular problem, but it might be unfit for another. ABS itself seems like it has dropped the support of Java backend based on the information in their documentation page [1] which does not include information about Java backend anymore. The toolchain required by both ABS Web Framework research [11, 12] also used an old version of ABS, which need to update to comply with newer features of ABS. Some newer features of ABS, e.g. `HttpCallable` annotation can be used to create a flexible web framework without the need to implement the web server logic in the target language generated by ABS. This feature reduces the number of artefacts that needed to be manage in the framework, thus ease the implementation of SPLE process in the web engineering. This fact simplifies the development of more SPLE-enabled frameworks that generate another programming language with minimal effort. With the potential ABS has as a modelling language to implement the SPLE process, the use of ABS can ease the implementation of the new SPLE-enabled web framework by model reuse mechanism.

Based on the need for more new research on SPLE-enabled web framework, we present an SPLE-enabled microservices web framework based on Erlang OTP. We also use microservices architecture with the structure defined by Fowler [8]. We also use ABS to create the framework, which makes it possible to use the already defined model from previous research [11, 12]. In this research, we use the Erlang backend of the ABS language based on the fact the capabilities the Erlang backend has. Erlang backend is one of the three backends provided by the ABS language and based on the information from

[1] https://abs-models.org/manual/#-abs-backends

the ABS documentation page with the most backend capabilities (four capabilities in total) in comparison to Maude (three capabilities) and Haskell (two capabilities). Since microservices are highly interoperable [10], we also use this architecture in our research. Microservices architecture enables integrating research that implements different technologies, thus making the research development more flexible.

This research conducted three process: literature study, framework design and implementation, and evaluation. We present a case study that demonstrates the functional parity of our framework with ABS Microservices Java [12] and compares the generated products resulted from both frameworks. We also use a load test that serves as functional test in demonstrating the functional parity and show the differences in runtime behaviour of the generated products. Then, we evaluate the framework by applying Six Quality Factor of Software Product Line implementation [14] to our research and ABS Microservices Java [12] as the comparison. Based on the result of the evaluation, we discuss the future research as the improvement to our framework.

This paper is ordered as follows: Section 1 provides the motivations illustration and a brief overview of the work. Section 2 explains the related research and the research method in developing an Erlang-based ABS Microservices Framework. Section 3 shows the detail on how the ABS Microservices Erlang Framework structure is implemented and the writer's proof of concept of the ideas. Section 4 presents the case study to illustrate the feasibility of the framework when implemented in a real-world problem. Section 5 discusses the evaluation of the implementation technique we used in this research. Section 6 and 7 conclude with conclusions and possible future works.

## 2. Related Research and Problem Overview

### 2.1. Related Research

The web application has become an essential part of many organizations, the research about web development has taken many researchers' interest. As an approach to creating software more effectively, the use of SPLE on web development has been researched, and one of the most exciting topics that arise is web application generator. Moreover, one of the most significant differences in developing applications using SPLE is the need to manage and design the commonality and variability of the application.

Some approaches are considered to manage the commonality and the variability of the web application. The industry usually adopts annotation-based approaches [15, 16], however majority of studies encourage the use of composition-based approaches [17, 18]. Kästner and Apel [18] then formulated the idea of combining both composition and annotative approaches. However, they only described general ideas for a combined approach and discuss the resulting characteristics (granularity, traceability, etc.). Then Horcas et al. [19] proposed to integrate annotations into a composition-based approach. Horcas et al. integrate a Common Variability Language (CVL) with annotations to manage variability in SPLE for web application [19]. Various artefacts must be generated to automatically generate a web application, such as HTML, JavaScript, CSS, JSON configuration file, and server-side source codes. By integrating CVL, a base model is used to represent various artefacts in the web application. The variability model in CVL will be used to declare the variation points within a base model that need to be reconfigured. Each variation point represented in the variability model will be mapped to one or more annotations in the artefacts. Later, the engine will compose the target application by processing the annotated artefacts according to the selected variations.

Another approach for generating web applications based on SPLE is delta-oriented programming (DOP) [20]. Abstract Behavioral Specification (ABS) [13] is one of the languages that implemented DOP. Our work is not the first research that has attempted to use ABS to create an SPLE-enabled web framework. Some research has been using ABS for generating web application [11, 12]. Aziz et al. [11] created an SPLE web framework based on the MVC design pattern. Naily et al. [12] use microservices architecture in their work of web framework instead of typical MVC. Naily et al. [12] and Aziz et al. [11] use the Java backend variant of ABS as opposed to the Erlang backend that we used. By adopting microservices, this framework can deliver a more flexible web application. Our research tries to create Erlang Microservices Web Framework using ABS. This framework will enable the developer to use more backend variants of ABS Microservices to satisfy a broader need. The developer can choose the target backend (Java or Erlang) based on the requirements defined.

### 2.2. Problem Overview

This subsection explains some concepts and challenges to developing the framework. There are some problems we need to solve in order to be able to create an SPLE-enabled web framework based on

microservices architecture. We explain this framework's implementation later in Section 3.

The main problem we need to solve to create the framework is related to how to implement the SPLE process in web application engineering. When developing an application with SPLE approach, there is a need for a specific mechanism that supports turning application requirements into code and derives application from the requirement. Web application aspects such as a mechanism to handle request and response, database connectivity, and non-feature application artefacts, such as configuration files and web server, need to be considered when designing the SPLE process. We also need to apply the SPLE process to the application's endpoints and database tables.

SPLE process is concerned about the diversity of application, commonality, and variability. According to [3], there are two engineering process, namely *Domain Engineering* and *Application Engineering*, in SPLE. Pohl et al. [3] defines *Domain Engineering* as a process to analyze the potential requirement of application (commonalities and variants). *Application Engineering* is the process of deriving a single variant tailored to the requirements of a specific customer from a software product line (SPL) based on the results of domain engineering. We will analyze the commonality and variability of application in domain engineering. Meanwhile, in application engineering, we will create the variant of applications/products from the requirement in the domain engineering.

We use ABS [13] to implement SPLE in this framework. In ABS, the domain engineering process is called feature modelling, and it is implemented in the feature model. The feature model defines all possible features selection that is valid. Meanwhile, in ABS, the domain implementation is implemented using the concept of DOP [20].

In DOP, SPL is defined as a core module and a set of delta modules [20]. Delta modules specify the changes to be applied to the core module in order to implement other products. A delta module can add classes to a product implementation or remove classes from a product implementation. Schaefer et al. [20] explain that in order to generate a product implementation for a particular feature configuration, the modifications of all delta modules with valid application conditions are incrementally applied to the core module. ABS implements the core module by creating a product variant with minimum functionality as a base product [13]. Hähnle [13] explains that the product variants with additional features are obtained from it by applying one or more deltas that realize the desired feature, implementing the delta module of DOP [20]. A microservices' module does not merely represent a feature. As ABS implement a feature by one or more delta modules [13], therefore the implementation of microservices is implemented by using delta modules. We define the base product in the core module and define some deltas to be applied to the core module to create product variants.

A web application consists of some parts that handle the entire process in the application based on its architecture. We adopt the design of microservices structure from [21]. Figure 1 shows that there are four layers in the design of microservices [21]. The resource layer acts as mappers between the application protocol exposed by the service and messages to the object representing the domain. The service layer contains logic implementation and coordinates across multiple domain activities. The domain layer represents a model or entity related to the microservice. The repository layer provides a collection of operations to access persistent data. A gateway encapsulates message passing with a remote service, marshalling requests and responses from and to domain objects. Except in the most trivial cases or when a service acts as an aggregator across resources owned by other services, a microservice will need to be able to persist objects from the domain between requests [21]. Usually, this is achieved using object relational mapping or more lightweight data mappers depending on the complexity of the persistence requirements. Often, this logic is encapsulated in a set of dedicated objects utilized by repositories from the domain.

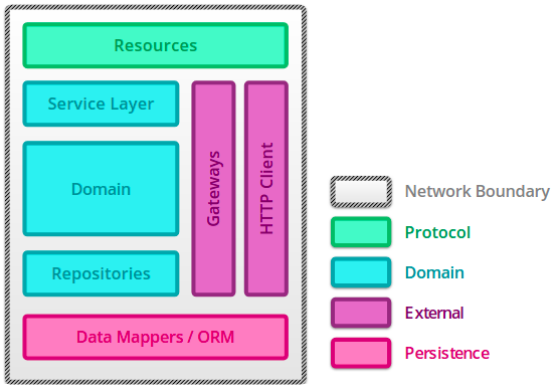


**Figure 1.** Microservices Structure [21]

The design from [8] is the guide we use to design and map the SPLE process to microservices web application structure. The design of the framework is shown by Figure 2. In general, there are four roles that the framework need to be fulfilled:

1) **Web Server**: The part of the application that controls how users access the web application, the request and the response.
2) **Microservices Modules**: The part of the web application in which the microservices structure is implemented. This is where the business logic is implemented.
3) **ORM**: Object Relational Mapping, the part of the application that connects the business logic to the database.
4) **Database**: The part that stores the data of the application.

From that design, we can define four different parts we need to define in our framework. We have the design of the application as shown by Figure 2. The core business logic lies in the microservices module. The core of the SPLE process will be applied to this module. Since we are using the Delta-Oriented Programming (DOP) approach, the microservices module should be divided into core and delta modules. This process can be handled easily by using ABS, as DOP is supported directly by the language. However, the microservices module does not only represent the application's feature. The microservices module should be the gateway to handle requests and responses as a web application. From the architecture defined in [8], this is the role of the resource layer. The resource layer acts as the request-response entity in the application. As ABS now support handling request and response natively, we explore the feasibility of features from ABS itself to build this layer. The newer version of ABS provide a support to access data with `HttpCallable` and `HttpName` annotations. Since our ABS will generate Erlang code, we also need to design the routing configuration from an Erlang file.

The implementation of the service layer and domain layer is straightforward. For domain and service, we treat the generated code from ABS as these layers without further changes. Service is a business logic layer of the application, so a standard Erlang code generated from ABS is sufficient to fulfil this role. A domain represents a model or entity and usually is tightly related to the data access layer (repository).

The challenge surfaced from the need to access the data from a data source like a database. ABS does not support access to the database natively, and the mechanism to access an external data source cannot be handled from modelling language as it will be tightly coupled to the generated target language; for this case, it should be handled by generated Erlang application. Another challenge that we need to solve to make the idea work is that ABS and Erlang use different programming paradigms. DOP, which ABS used, is still utilising objects, similar to Object-Oriented Programming (OOP), and in the opposite, Erlang fully utilises functional programming. The model that we need to connect to the database is defined in ABS, and the architecture from [8] also make sense in language that respects the use of object. Object Relational Mapping (ORM) is used to map from object to the relational database to ensure flexible connection to the database. We decide to use external libraries to support the creation of the ORM. Unfortunately, no approach is available as far as we know to map from object to functional programming style that is used in Erlang easily. The class fields of ABS need to be converted manually as a parameter to the function in Erlang. We create `orm.erl` to comply with the role of the ORM in generated Erlang code. The database query such as `select, insert, update, delete` will be controlled by the `orm.erl` that utilised the external library to access the database. Since we use external libraries after the code is generated, no mechanism is designed to create the database automatically. Also, there is a need to modify the generated Erlang code to use the `orm.erl`.

A web application has other types of artefacts besides the application logic files that need to exist in order for the web application to work. Beside the `orm.erl` file, there are artefacts like configuration files, build script and external library dependencies that need to be considered and managed by the SPLE process. These artefacts can be treated as mandatory requirements in the domain analysis as every product in the product line will need its existence. However, as these files are not directly related to the feature, they can be added later after the core SPLE process. We will present the design that tackles the problem overview in the next section.

## 3. The ABS Microservices Erlang Framework

In this section, we explain the implementation of ABS Microservices Erlang. First, we show the framework's workflow and how to add new features in case of a new requirement. Then, we present the framework's structure based on the previous section's design. Later, we explain the build process in the framework.

### 3.1. Framework Workflow

There are seven processes in our proposed framework. The starting process might be the initialisation of the framework or an addition of a new product definition. The initialisation is the beginning of the

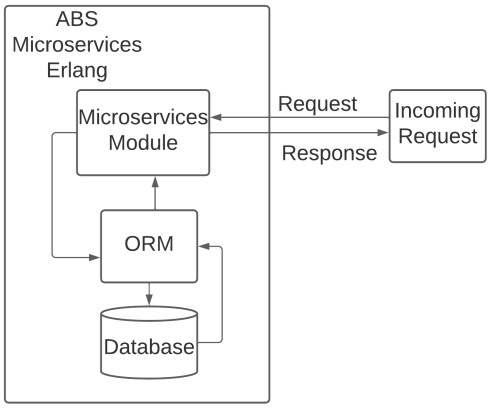


**Figure 2.** ABS Microservices Erlang Design

framework workflow where the initial features and product are analysed. The result of the initialisation process will be the input of the following process in the workflow. The new product definition can be added throughout the framework life cycle as the product lines grow and more new products are added. Figure 3 shown the framework' workflow process.

The workflow process begins with identifying features requirements. SPLE approach aims to create multiple products instead of only one in the traditional approach. This process identifies and analyses all the product' requirements. These requirements will then be turned into a mapping of the product and the list of features it contains, and the relation between features. A feature might be related to another feature, leading to a need to select both features. All of these requirements' identification will be the input of the second process.

The second process defines the ABS model representing a product, and the third process defines the deltas to derive the other product variants. The second and third processes are related to the core and the delta module concept of the DOP. Other product variants will be derived from the core module by applying deltas in the third process. The delta contains the modification needed to derive that features. The different feature needs different delta. There might be a feature that needs another feature resulting from another delta. So the other product variants will be contained one or more deltas according to what feature they contain. To make sure no conflicts result from this process, DOP introduces the concept of delta ordering [20]. The delta ordering ensures the order of the delta by using `after` clause to ensure the features that delta needs exist. Then, `when` clause is used when a feature that needs that delta is selected.

The fourth process defines the product lines and the products they contain. Then, all products will be defined based on what features they contain. In this process, the feature model and the products are defined. The feature model is created from the feature model, resulting from the first process. The feature model defines all the features, whether they are mandatory or optional, and their relationship with other features. The products list all the features that the product contains. If the features listed by that product require a delta, the delta will be applied based on the delta definition from the previous process.

The fifth process is the product derivation process. After all the models are and the products are defined, we can use the framework to generate products. After the product generation is completed, there is a need to modify some configuration files and web artefacts before the product can be used as the final process in the workflow. The detailed process to create a product variant and the modification needed will be explained further in the following subsections.

A new product can be added to the framework. The product addition can be done by adding the new products to the product definition. New deltas need to be defined if the new product needs new features. The new product then can be generated the same way as the other product that has been defined initially.

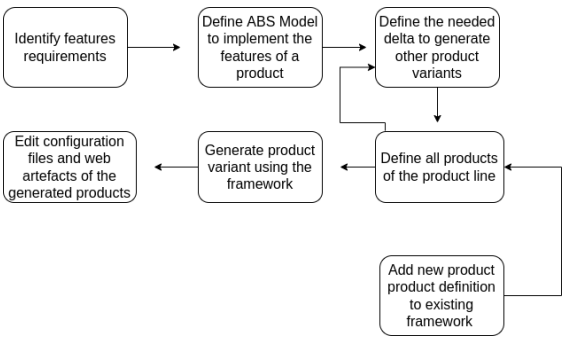


**Figure 3.** ABS Microservices Erlang Workflow

There are two ways to add a new feature, as shown by Figure 4. First, we analyse the significance of the new feature. If most products require this new feature, we implement the new feature as the ABS model, similar to the initial process. On the other hand, if the new feature is only required by some specific products, the new feature will be implemented in the required products. In this case, the feature will be implemented manually as Erlang code.

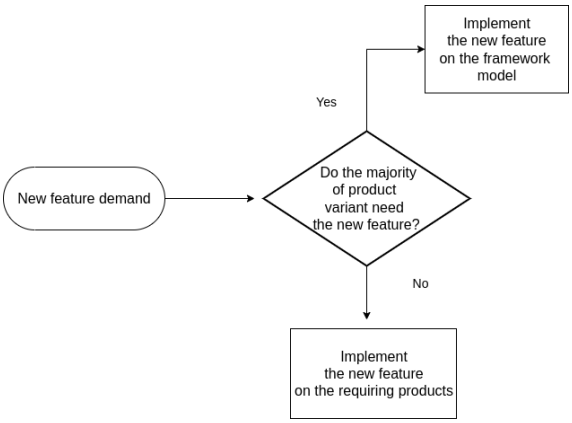


**Figure 4.** ABS Microservices Erlang New Feature Addition Workflow

## 3.2. Framework Structure

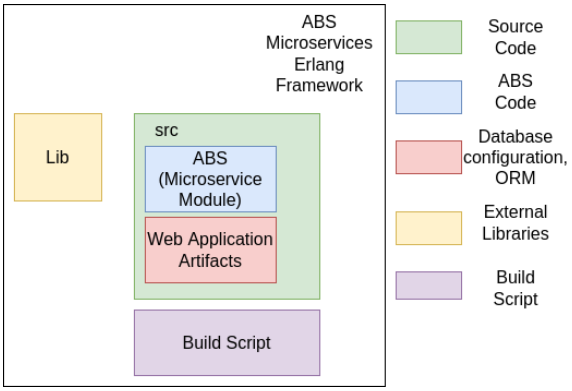


**Figure 5.** ABS Microservices Erlang Directory Structure

We design a framework structure based on the problem overview discussed, as can be seen in Figure 5. In general, there are four types of artefacts in this framework: microservices module, Web application artefacts, build scripts, and external dependencies. The microservices module consists of the features that the application owns by the product line. Web application artefacts are non-feature artefacts related to the web application process, e.g., Object Relational Mapping (ORM) and database configuration. External dependencies to run web artefacts are stored in the `lib` directory. Build scripts are Bash and Python scripts that will call ABS compiler and move the related files to the generated product application.

The microservices module is implemented using ABS. Because all the features defined in the product line reside in this module, all business logic source codes are implemented within the microservices module. Consequently, the primary SPLE process should be applied to this module, and the mandatory non-feature artefacts will be added after the code generation.

In order to implement the SPLE process in the microservices module, the ABS source codes are divided into core and delta modules. The core module comprises ABS modules for implementing the base framework functionality and the mandatory features of the application. It is organised into a set of directories in the file system, where each directory represents a layer in the architecture. The core module also contains a directory containing the ABS modules related to the framework and the SPLE process, such as the ABS modules for HTTP request routing, feature model, product line configuration, and product specification.

The delta module comprises ABS deltas for transforming the core product of the application into a variant that contains the desired features. It mirrors the core module structure where each directory in the module contains the ABS deltas for the corresponding layer in the architecture. The delta module also contains a directory containing the ABS deltas for customising the framework. For example, the HTTP request routing for a feature can be updated by creating an ABS delta that modifies the routing scheme defined in the core module.

In addition to the core and delta modules, the framework contains an Erlang template file for the ORM, vendorised Erlang libraries for database and HTTP server functionalities, and the shell scripts used by the product selection process. Figure 5 represent these files as the non-feature web artefacts and external libraries. Non-feature artefacts, such as configuration files and ORM, will be placed outside the source code folder (outside `src`). The code generation process then moves the needed files to generated product application. While this approach seems to be enough to make the generated product application works, we also need to consider that each generated product application might need a different application configuration. Application configuration such as database connection should be different on each generated application. We design a generation process for the configuration files to comply with this need. As at the current state, our framework runs the generated product on the same machine as the code generator, and reading configuration environment variables would be complex as there might be plenty of generated products in the machine. This complexity is why we use generated configuration files instead of copying the configuration that read from the environment variables. While the result is hardcoded configuration, for the state of generated

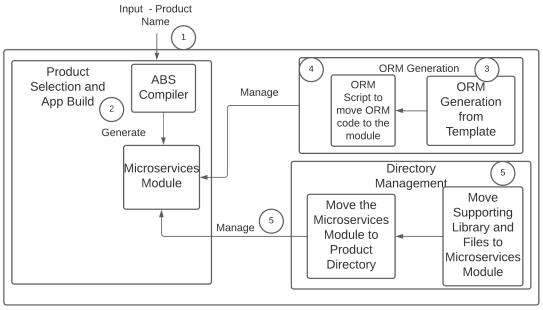


**Figure 6.** ABS Microservices Erlang Build Process

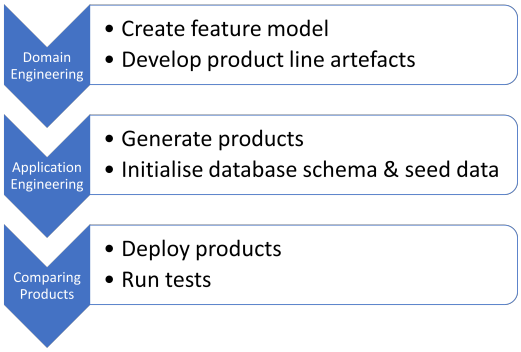


**Figure 7.** Case Study Stages

product applications resided in the same machine as the framework, we do not need to manage the variables of the generated products. There is also a directory for storing SQL scripts for initialising the schema and the seed data in the database. However, the framework does not support automated execution of the SQL scripts yet. Therefore, the database schema and the seed data must be initialised manually.

### 3.3. Build Process

The ABS Microservices Erlang build process consists of three parts: Product Selection, ORM Generation, and Directory Management. We made a build system using a Bash script to do the process.

The first step of the build process is to select and create products. The build script accepts an argument `p` that represents the product name. The value of `p` will be used as another argument for the code generator in ABS to select and create products. The result of this product creation from ABS will be put in a folder named `gen`. We now have finished step 2 of Figure 6 and finished the first part of the build process.

The next part is about ORM Generation. We need to connect the generated application to a database and update the generated ORM code. The previous ABS Microservice, ABS Microservice Java [12], uses PostgreSQL as its database. We need to use PostgreSQL also in ABS Microservice Erlang. In order to use PostgreSQL, we use a library, `epgsql`, to connect ABS Microservice Erlang to PostgreSQL Database.

ORM is needed to convert data stored in relational database tables into objects that we can use in our application. ABS Microservice Erlang does not have ORM initially, so we also need to adjust ABS Microservice Erlang to use *epgsql* functions as ORM on the generated repository layer. We made a template file, `orm.erl`, that consists of database-related functions such as connection setup and basic query functions using `epgsql`. Our build script does this on step 3 of Figure 6.

The supporting libraries and `orm.erl` are moved to the generated application, as seen in step 4 of Figure 6. The last step is to create product folders according to the input `p`. This is done with a Python script.

All model objects that have been annotated with `HttpCallable` and `HttpName` can be accessed at `[HOST]/call/<Object>`. ABS Microservice Erlang sets the `/call` route by default. We can reconfigure the route on `modelapi_v2.erl`, but for now, we leave it as is. After all steps have been done, the application is ready to run.

## 4. Case Study

This section will describe a case study of a microservices-based project built using the ABS Microservices Erlang framework. We outline the process of implementing the case study from domain engineering to application engineering phase according to the SPLE methodology, as depicted at Figure 7. We develop the required artefacts and generate the products using both ABS Microservices Erlang and Java frameworks. Then, we deploy the generated products from both frameworks on a running server and perform comparison on aspects such as framework usability, functional parity of the generated products, and memory usage of the generated products.

### 4.1. Domain Engineering

We start by defining the domain of the product line and creating the representative feature model. We choose a charity organization system domain analyzed by Laily et al. [22] as an example. The

feature model describes the features that will be mandatory and optional across every possible product in the product line. We follow the $\mu$TVL syntax in ABS language to create the feature model in textual format.

The next step is developing the core module of the application. The core module consists of ABS modules that implement all the mandatory features defined in the feature model. The implementation also needs to follow the convention dictated by the ABS Microservices Erlang framework. For example, the HTTP endpoint to a particular feature needs to be defined in an ABS module provided by the framework.

Once done creating the core module, we start developing the delta module. The delta module contains the ABS deltas that will be used for modifying the core module when certain features are required. One or more ABS deltas from the delta module can be applied during product generation. For instance, if a new feature requires an HTTP endpoint not yet available in the core module, one can create a delta module with an ABS delta that will modify the existing HTTP router by adding the new HTTP endpoint definition.

Finally, we define the product line configuration and the product specification. The product line configuration contains the list of features and deltas in the product line. It also contains the constraints that describe the relations among the features and the deltas. The product specification contains the list of products that the product line can generate. Each product describes the features available in the product, both the mandatory features and the optional features.

The same ABS artefacts produced in this phase can be reused in a product line built on the ABS Microservices Java framework. However, the artefacts need to be adjusted due to the difference in how the ABS Microservices Java framework provides an HTTP server. The details are discussed in section 3.2.

## 4.2. Application Engineering

First, we set up the toolchain for performing the product selection in the SPLE process. The toolchain requires a specific version of ABS tools [2], Erlang OTP 22, Java 8, Python 3, and Bash shell. ABS tools provide a compiler that parses ABS modules and generates the source code written in a supported language such as Erlang or Java. In addition, ABS tools also include a model checker to verify the correctness of the product line.

Second, we prepare the generated product's database schema and seed data. As mentioned in section 3.2, the database schema and the seed data are written in SQL scripts and executed outside of the framework. Once the database is set up, we create a JSON file containing the generated product's database connection configuration. The framework will parse the JSON file using a Python script during product generation and use the parsed values to fill in the ORM template file required by the generated product.

The framework provides a Bash shell script that accepts a parameter containing the desired product name to generate a product. The Bash shell script executes a series of commands and a Python script that invoke the ABS tools to generate the source code artefact. In addition, the shell script also sets up the database and organizes the resulting artefact into a new directory named after the product.

## 4.3. Comparing Generated Erlang Product with Java

We set up a virtual machine (VM) running GNU/Linux-based operating system on a cloud service provider. The VM contains the required toolchain for building the product line and running the generated product from both ABS Microservices Framework. Then, we create a product with the same features using both ABS Microservices Framework and deploy the products on separate ports. Each product is deployed and run using the default configuration of each respective runtime, which is Java Virtual Machine (JVM) and Erlang runtime system (`erl`).

To simulate user traffic to the running products, we use Apache JMeter [3] to perform two load tests on each product. The load tests are executed from an Internet-connected personal computer (PC). Each load test simulates concurrent users repeatedly accessing an HTTP endpoint for 30 seconds with 10 and 100 simulated users. Additionally, each load test also serves as functional test that verifies the HTTP response from each product during testing. The tests expect that all responses from each product must contain a valid JSON data.

During each load test to each product, we measure the memory usage of both products by using `ps_mem` [4] on the VM. We measure the memory usage before the load test starts and when the load test simulates the peak number of concurrent users. Table 1 summarizes the memory usage of each product from each load test. We can see that the product

[2]https://github.com/abstools/abstools/tree/797bb73329ba

[3]https://jmeter.apache.org/

[4]https://github.com/pixelb/ps_mem

**Table 1.** Memory Usage

| Product | Number of Simulated Users | Idle Traffic | Peak Traffic |
|---|---|---|---|
| Java | 10 | 201.6 MiB | 275.9 MiB |
| | 100 | 199.2 MiB | 279.4 MiB |
| Erlang | 10 | 23.6 MiB | 27.6 MiB |
| | 100 | 22.4 MiB | 27.3 MiB |

that runs on Erlang runtime consumed less memory than Java.

### 4.4. Discussion

We have demonstrated that similar ABS modules can be used both in ABS Microservices Erlang and ABS Microservices Java frameworks to build a product line. Additionally, we also showed that both frameworks could produce two equivalent products with the same features. There is a minor difference in how to implement the ABS modules that are responsible for representing the HTTP API endpoints. We use `HttpCallable` and `HttpName` annotations on ABS modules that represent the API endpoints. The Erlang code generator uses the annotations in the ABS toolchain to bind the generated Erlang code with the built-in HTTP server in the Erlang runtime. Meanwhile, the ABS Microservices Java does not use annotations for wiring up the application with the HTTP server. The ABS Microservices Java was developed on an older version of the ABS toolchain where the HTTP-related annotations were not available yet. It uses a custom Apache Tomcat-based HTTP server that integrates with the generated Java code through the FLI (Foreign Language Interface) mechanism in ABS language.

We observed a much smaller memory footprint on the product than run on Erlang runtime in terms of memory usage. The memory consumption of Erlang-based products is roughly ten times lesser than its Java counterpart. This lesser memory consumption leads to a possible operational cost reduction when deploying a product on a cloud environment, where the costs are charged based on the utilization of available computing resources. If there is a concern on cost-effectiveness when running products on the cloud, one could prefer to build and deploy a product using ABS Microservices Erlang instead of ABS Microservices Java.

## 5. Evaluation

This section explains the evaluation criteria we use to examine the product line implementation technique we use. First, we explain the metrics to evaluate the framework implementation and the evaluation of our framework and also ABS Microservices Java from Naily et al. [12] as a comparison. Then, we discuss the evaluation result and summarize the evaluation.

Different implementation techniques for product-line development have different characteristics and mutual strengths and weaknesses [14]. Apel et al. [14] present six quality criteria to access tradeoffs and compare product line implementation strategy. Apel et al. introduced these quality criteria as the quality criteria which product line implementation should ideally meet:

1) **Preplanning Effort**: The effort needed to prepare an implementation technique.
2) **Feature Traceability**: The ability to track a feature from the problem space into the solution space.
3) **Separation of Concerns**: The separation of features both in design and code, such that the relationship
between features and corresponding design and implementation's artefacts are explicit.
4) **Information Hiding**: To decompose a system into modules and to divide each module into an internal and an external part.
5) **Granularity**: The granularity of features. A level of granularity refers here to the hierarchical structure of an implementation artefact,
typically, defined by a containment relation among the artefacts' structural elements.
6) **Uniformity**: All features should be encoded and synthesized in a similar
manner.

The second to sixth quality criteria are the quality criteria that check whether implementation criteria satisfy the defined quality. Meanwhile, the first quality criteria objective is to show the effort to preplan implementation criteria and present the implementation technique's complexity. As the first quality criteria defined by Apel et al. yet to give the explicit definition of the complexity, we extend this quality factor with three categories of complexity based on the effort needed to derive a new product to the product line using that implementation technique. We also design a complexity modifier specific to the web application to assess a software product line implementation to these three categories

As shown by Table 2, there are six complexity modifier. Their value is weighted based on their significance on the resulting product. A high effort and significant impact on the generated product resulted in a high value on the modifier. Every single complexity modifier is explained below:

**Table 2.** Pre-planning Effort Complexity Modifier

| Complexity Modifier | Value |
|---|---|
| Customised Compiler | 2 |
| Configuration files modification after product generation | 1 |
| Feature artefacts modification after product generation | 3 |
| External or customised middleware for URL routing | 1 |
| Tools forward incompatibility | 1 |
| Web Artefacts modification after product generation | 3 |

1) **Customised Compiler (C1)**: The use of a customised compiler.
   This implied a use of non-standard compiler to applying a certain process.
2) **Configuration files modification after product generation (C2)**: There is a need to change the configuration files after the product generation before the product can be worked.
3) **Feature artefacts modification after product generation (C3)**: There is a need to edit the feature related artefacts/logic before the product can be used.
4) **External or customised middleware for URL routing (C4)**:External libraries or artefacts are needed to configure the routing and the request-response of the web application.
5) **Tools forward incompatibility (C5)**: Depicts the framework's inability to update the tools to use the latest technology stack in case of deprecation.
6) **Web Artefacts modification after product generation (C6)**: There is a need to update web application related artefacts, such as database connectors, before the product can be used.

There are two complexity modifiers with value weighed at three: feature artefacts modification after product generation and web artefacts modification after product generation. These two complexity modifiers need a high effort, and the generated product cannot be used before both are completed. A customised compiler is rated two as the effort to create the compiler is high, but afterwards, it can generate multiple products without a need for modification after the generation process. This complexity modifier is related to tools forward incompatibility as a customised compiler makes it hard to upgrade later. The rest of the complexity modifiers are rated one as they do not need a high effort, or the product can be generated without fulfilling that complexity modifier. This value then will be used to determine the pre-plan complexity. The metrics is shown in Table 3.

**Table 3.** Pre-planning Effort Complexity

| Pre-planning Effort Complexity | Pre-planning Effort Complexity |
|---|---|
| Low Complexity | 0-3 |
| Medium Complexity | 4-6 |
| High Complexity | Larger than 6 |

A low complexity implies that adding a new feature only slightly change or better does not need to modify the existing codebase. Low complexity is the ideal complexity that needs to be satisfied by an implementation technique. The existing reusable artefacts can be used to derive a new feature and product. The difference with the low complexity criteria is that the change needed in the low complexity is mainly on using the reusable artefacts to derive product, e.g., changing the list of product and what artefacts that product use. Meanwhile, medium criteria complexity implies a need to change the existing codebase or the resulting product to use the reusable artefacts as the working product. High complexity denotes an inflexible implementation technique, such that there are many changes needed to add a new feature or derive a product that does not exist initially.

The result of the pre-plan complexity comparison between our research and ABS Microservices is shown by Table 4. Our framework has a complexity modifier of 7, categorised as high, and on the other hand, ABS Microservices Java has the value of 5, categorised as medium. ABS Microservices Erlang that we developed still needs modifications after the products are generated. Not only do we need to modify the generated products' configuration files, but there is also a need to modify web artefacts such as database connection in the repository layer. The need for these modifications leads to more complexity in the product generation process for our framework. On the other hand, ABS Microservices Java does not possess this problem, but instead, it has a problem with the maintainability and sustainability of the framework. It will not be easy to update the tools to use newer technology in case of deprecation because of a need to update the customised compiler. ABS Microservices Java also has the problem of the framework complexity because a lot of the middleware such as URL routing and request-response management uses external libraries.

Table 5 shows the overall evaluation comparison of both frameworks using the quality criteria defined by Apel et al. [14]. The quality criteria of Feature

**Table 4.** Pre-Planning Effort Complexity Comparison

| Framework | Complexity Modifier Value | Pre-Planning Effort Complexity | Complexity Modifier |
|---|---|---|---|
| Erlang-based | 7 | High | C2,C3, C6 |
| Java-based | 5 | Medium | C1,C2, C4, C5 |

**Table 5.** Evaluation Comparison of Erlang-based and Java-based [12] ABS Microservices

| No | Quality Criteria | Erlang-based | Java-Based |
|---|---|---|---|
| 1 | Preplanning Effort | High | Medium |
| 2 | Feature Traceability | V | V |
| 3 | Separation of Concerns | V | V |
| 4 | Information Hiding | V | V |
| 5 | Granularity | V | V |
| 6 | Uniformity | V | V |

Traceability, Information Hiding, and Granularity have been handled indirectly by ABS, which is used by both frameworks. ABS is a language that naturally supports the SPLE process, which eases the development of the SPLE-enabled framework. SPLE process in ABS is implemented by Delta-Oriented Programming (DOP) approach. A feature diagram used to represent the features of the product line is translated into ABS codes. The ABS code version of the feature diagram lists the product line's features and validates the feature selection in the product derivation. ABS also has a module system that helps to represent the relationship between artefacts via export and import. This module system complies with the Information Hiding quality criteria since the module system explicitly states the dependencies of that module and what entity the module exposes. As there are explicit dependencies among modules, Granularity quality criteria are also fulfilled as it gives the ability to track the relationship between modules as reusable artefacts. We can track the relationship between artefacts, e.g. dependencies between artefacts and combine it with DOP by using this mechanism. We can trace features from problem space to the solution space, hence complying with Feature Traceability criteria.

Both frameworks satisfy Separation of Concerns and Uniformity by integrating the SPLE process of ABS and microservices design from Fowler [8]. A feature is defined by the cooperation of all four layers of the microservices design. So the SPLE process needs to be applied to all layers of microservices. The application of the SPLE process denotes a pattern to add a new feature, hence maintaining the consistency asked by uniformity quality criteria. Furthermore, since all the layers have their explicit role and the DOP process in the ABS also split the module based on their role (core or delta), there is a clear separation of concerns.

The significant difference between both frameworks is how the framework handles the web engineering process; hence, these frameworks have different complexity. Both frameworks use ABS compiler to generate product in the target language, but ABS Microservices Java [12] uses a customised build chain as opposed to directly using ABS compiler as we did in this research. The information needed to build web application artefacts such as Object Relational Mapping (ORM) and database tables is inferred from the abstract syntax tree during build time. As opposed to that approach, our framework split the process to generate the product's feature and web application artefacts. This approach has an advantage over merging the process as one extensive build process because different processes imply it will be easier to interchange the web application artefacts and technology since the two processes are independent. We can choose the library and technology used to build the web application artefacts without affecting the feature build process. The use of a custom build chain implies the specific technology needed for the product application to work as intended. Since we are not embedding any specific process into the build process of the ABS compiler, the future update of the compiler would not disrupt the framework as much as it did to the renewal of ABS Microservices Java. If there is a need to update the web component, for the web artefacts are in a different process from the build, they can be replaced without affecting the primary ABS build process.

More effort is needed to change the technology used by ABS Microservices Java to build web application artefacts. For example, ABS Microservices Java uses PostgreSQL as the database management system. Since the `create table` statement read from abstract syntax tree and then directly written as PostgreSQL dialect, to change the database management system to, for example, MySQL, there is an effort needed to change the database table generator to comply with MySQL dialect. In comparison, our ABS Microservices Erlang framework is more flexible on web application artefacts technology. The needed non-feature artefacts are added later after the product derivation process, so it does not depend on the build process. If there is a need to change web application artefacts technology, the new technology can replace the old technology in the library directory.

However, as shown by Table 5, our approach has an overhead in the complexity of preplanning effort in comparison to ABS Microservices Java [12]. Our ABS Microservices Erlang does not read the infor-

mation from the abstract syntax tree. Consequently, the generator does not have all the needed information to generate a non-feature artefact related to the feature, e.g. database access from the repository layer. The consequence is that there is a need to modify the resulting product application code to be working as intended. This is not the case with ABS Microservices Java. Overall, the preplanning complexity of ABS Microservices Erlang is higher than Java-based.

## 6. Conclusion

This research presents an ABS Microservices Erlang, a web framework based on the SPLE concept. The ABS Microservices Erlang framework is able to semi-automatically generate a web application based on delta-oriented programming [20]. ABS is used to develop a web application using the framework that a standard library supports. The framework also enables the further adoption of SPLE approaches in Web application development. Both frameworks use the ABS to model the features and the application. The same ABS files can be reused to generate a microservices web application with the preferred language. We observed a smaller memory footprint on the generated products made by the ABS Microservices Erlang framework based on our profiling process compared to ABS Microservices Java. The result may help the user decide on which framework to use in a memory-constrained environment. We also evaluated our framework implementation technique and ABS Microservices Java as the comparison. The result shows that there is still a need to work on our framework's preplanning effort quality criteria. Our approach that splits the process of the main ABS compiler build and the installation of web application components promises better maintainability of the code, should there is a significant update to the ABS compiler. However, the product generation in our framework has yet to achieve complete automation, which ABS Microservices Java has accomplished, thus increasing the preplanning effort.

## 7. Future Research

Work is needed to reduce the preplanning effort in ABS Microservices Erlang by applying some automation in the web application artefacts generation and installation process. While we can automate the process of generating the ORM, there is a manual step involved where we have to insert the corresponding function call in the generated code to perform query or data manipulation. For example, we have to add `orm:findAll` in a function that returns the collection of objects from a table. This manual step implies a need to add new SQL statements every time there is a new feature or entity. A mechanism to read entities information from ABS's abstract syntax tree can be considered to improve the automation ability of the framework. In addition, subsequence research can further investigate improving the code generation process from ABS ORM modules into generated Erlang code.

## References

[1] I. Schaefer, R. Rabiser, D. Clarke, L. Bettini, D. Benavides, G. Botterweck, A. Pathak, S. Trujillo, and K. Villela, "Softw. diversity: state of the art and perspectives," *Int. J. on Softw. Tools for Technol. Transfer*, vol. 14, no. 5, pp. 477–495, 10 2012, copyright - Springer-Verlag 2012; Last updated - 2014-08-30.

[2] D. Di Ruscio, M. Chechik, and B. Rumpe, "9th workshop on model. in softw. eng. 2017," in *2017 IEEE/ACM 9th Int. Workshop on Model. in Softw. Eng. (MiSE)*, May 2017, pp. 1–1.

[3] K. Pohl, G. Bockle, and F. van der Linden, *Softw. Product Line Eng.: Found., Princ., and Techn.* Berlin: Springer-Verlag, 2005.

[4] A. Miniukovich and A. De Angeli, "Webpage aesthetics: One size doesn't fit all," in *Proc. of the 9th Nordic Conf. on Human-Comput. Interaction*, ser. NordiCHI '16. New York, NY, USA: Association for Computing Machinery, 2016.

[5] H. Keshavarz and M. E. Givi, "Website evaluation frameworks: Is oriented vs. business oriented models," in *2020 6th Int. Conf. on Web Research (ICWR)*, 2020, pp. 223–228.

[6] V. F. de Santana and M. C. C. Baranauskas, "Continuous web personalization using selector-template pairs," in *Proc. of the 16th Int. Web for All Conf.*, ser. W4A '19. New York, NY, USA: Association for Computing Machinery, 2019.

[7] D. Yadav, A. Shinde, A. Nair, Y. Patil, and S. Kanchan, "Enhancing data security in cloud using blockchain," in *2020 4th Int. Conf. on Intell. Comput. and Control Syst.*, May 2020, pp. 753–757.

[8] M. Fowler, "Microservices guide," 2019. [Online]. Available: https://martinfowler.com/microservices/

[9] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, *Microservices: Yesterday, Today,*

*and Tomorrow*. Cham: Springer International Publishing, 2017, pp. 195–216. [Online]. Available: https://doi.org/10.1007/978-3-319-67425-4_12

[10] M. A. Jarwar, S. Ali, M. G. Kibria, S. Kumar, and I. Chong, "Exploiting interoperable microservices in web objects enabled internet of things," *2017 Ninth International Conference on Ubiquitous and Future Networks (ICUFN)*, pp. 49–54, 2017.

[11] A. Aziz, M. R. A. Setyautami, and A. Azurat, "A web-based softw. product line eng. framework," in *2019 Int. Conf. on Adv. Comput. Sci. and inf. Syst.* IEEE, Oct 2019, pp. 21–26.

[12] M. A. Naily, M. R. A. Setyautami, R. Muschevici, and A. Azurat, "A framework for model. variable microservices as softw. product lines," in *Lecture Notes in Comput. Sci.*, vol. 10729 LNCS, 2018.

[13] R. Hähnle, "The abstract behavioral specification language: A tutorial introduction," in *Formal Methods for Components and Objects: 11th Int. Symposium, FMCO 2012, Bertinoro, Italy, September 24-28, 2012, Revised Lectures*, E. Giachino, R. Hähnle, F. S. de Boer, and M. M. Bonsangue, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 1–37.

[14] S. Apel, D. Batory, C. Kästner, and G. Saake, *Feature-Oriented Softw. Product Lines*. Berlin: Springer-Verlag, 2013.

[15] C. Hunsen, B. Zhang, J. Siegmund, C. Kästner, O. Leßenich, M. Becker, and S. Apel, "Preprocessor-based variability in open-source and industrial softw. syst.: An empirical study," *Empirical Softw. Eng. 2015 21:2*, vol. 21, pp. 449–482, 4 2015.

[16] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in softw. product lines," in *2008 ACM/IEEE 30th Int. Conf. on Softw. Eng.*, 2008, pp. 311–320.

[17] F. Benduhn, R. Schröter, A. Kenner, C. Kruczek, T. Leich, and G. ANDSAAKE, "Migration from annotation-based to composition-based product lines: towards a tool-driven process," in *Proc. Conf. Advances and Trends in Softw. Eng. (SOFTENG). IARIA*, 2016, pp. 102–109.

[18] C. Kästner and S. Apel, "Integrating compositional and annotative approaches for product line eng." 2008.

[19] J.-M. Horcas, A. Cortiñas, L. Fuentes, and M. R. Luaces, "Integrating the common variability language with multilanguage annotations for web eng." 2018.

[20] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella, "Delta-oriented programming of softw. product lines," in *Softw. Product Lines: Going Beyond*, J. Bosch and J. Lee, Eds. Heidelberg: Springer Berlin, 2010, pp. 77–91.

[21] T. Clemson, "Testing strategies in a microservice architecture," 2014. [Online]. Available: https://martinfowler.com/articles/microservice-testing/

[22] I. L. Laily, O. Komarudin, S. Fadhilah, and A. Azurat, "Progressive learning design strategy to improve impact maturity of charity organizations," in *2018 Int. Conf. on Adv. Comput. Sci. and Inf. Syst.* IEEE, Oct 2018, pp. 39–44.