Vigilada Mineducación

AN APPROACH TO IMPLEMENT SPL COMPOSED OF INTERCONNECTED APPLICATIONS AND TO DEPLOY THEM TO THE CLOUD

Un enfoque para implementar LPS compuestas por aplicaciones interconectadas y desplegarlas en la nube

VERONICA LONDONO OSORIO

Trabajo de grado

Asesor(es)
Daniel Correa Botero
Paola Andrea Vallejo Correa

UNIVERSIDAD EAFIT
ESCUELA DE INGENIERÍAS
MAESTRÍA EN INGENIERÍA
MEDELLÍN

2023

An Approach to Implement SPL Composed of Interconnected Applications and to Deploy them to the Cloud

Abstract. Software product lines (SPL) are a systematic reuse technique that both academy and industry have been using in recent years. The main idea is to generate different software products through the reuse of a set of assets. Different authors have proposed different approaches and techniques to the construction and maintenance of these assets. However, most of these approaches are designed to support the development of standalone applications, and there is not support for a product deployment. In a previous work, we developed fragment-oriented programming (FragOP), which is a framework used to design, implement, and reuse SPL assets. And a tool called VariaMos which supports FragOP. In this work, we enhanced VariaMos and FragOP to support the definition of SPL composed of interconnected applications and automate the deployment of the generated applications to the Cloud. Finally, we developed a running example (a ToDo SPL) to show some preliminary results of the new approach.

Keywords: software product lines, product deployment, fragment-oriented programming, interconnected applications.

1 Introduction

A software product line (SPL) is a collection of similar programs that satisfy a particular market segment's needs and are developed from a common set of core assets [1]. The core assets consist of a common code base (such as components) and variants. The construction of these components and variants is the key to an efficient SPL product derivation (which consists in generating specific software products based on the SPL core assets). There have been multiple component implementation approaches and tools, such as FOP, FragOP, Antenna, DeltaJ, AHEAD, and CIDE [2]. However, most of these approaches only support the derivation of self-contained applications. These applications are derived as a single product containing everything they need to work.

Nevertheless, many software applications are developed as more complex projects with interconnected applications. For example, an application that consists of a front-end application interconnected with a back-end application. Besides, deploying these derived applications (on cloud servers) is a manual task that dismisses the SPL benefits, such as accelerating time-to-market, improving product quality, and reducing costs.

For example, Casquina & Montecchi [3] have identified a lack of integration between variability realization mechanisms and version control systems (which can be used to automate the deployment) that reduces SPL attractiveness in the software development industry.

In previous work, we developed an SPL implementation approach called Fragment-oriented programming (FragOP) and a tool that supports this approach called VariaMos [4]. FragOP and VariaMos provide capabilities to define and implement SPLs and derivate software products. In this paper, we enhanced the FragOP and VariaMos capabilities to (i) support the definition of SPL composed of interconnected applications and (ii) automate the deployment of the generated applications to the Cloud.

The support to define SPL composed of interconnected applications allows developers to define and manage more complex projects inside a single SPL. Such as managing a front-end application containing the user interface of a specific domain and managing a back-end application that contains the logic and the data persistence of the same domain. The front-end application will need a connection to the back-end application, and both applications should be derived as different products.

The automated deployment allows developers to connect the derived applications with platforms such as GitHub and cloud servers (such as the Google Cloud Platform) to automate the deployment process.

The rest of this paper is structured as follows. Section 2 presents and recaps the FragOP approach and the VariaMos tool. In section 3, we present a running example that will allow us to exemplify and explain the new proposal in a practical way. In section 4, we present and explain the new proposal. In section 5, we show some deployment results. In section 6, we discuss the related work. Finally, section 7 summarizes the contributions and presents future research directions.

2 Fragment-oriented programming and VariaMos

2.1 Fragment-oriented programming (FragOP)

Fragment-oriented programming (FragOP) is a framework used to design, implement, and reuse domain components in the context of an SPL [5]. FragOP is based on the definition of six fundamental elements: (i) domain components, (ii) domain files, (iii) fragmentations points, (iv) fragments, (v) customization points, and (vi) customization files. The fragments act as composable units (compositional approach), and the fragmentation points act as annotations (annotative approach). This mix of compositional and annotative approaches allows the FragOP to support multiple assets implemented over different languages, such as PHP, Java, JSP, CSS, HTML, and JavaScript. In addition, FragOP has two main capabilities: assembling and customization of components [6].

We will only explain a few FragOP elements and capabilities in detail because they were presented in previous work [6]. However, we will make a quick recap of some of them. Figure 1 shows the assembling capabilities of FragOP. In this case, (1) a domain file (*header.jsp*) supports the code variability (2) through the inclusion of a fragmentation point (*menu-modificator*). Additionally, (3) a fragment (*alterHeader.frag*) specifies (4) a code alteration (to include a new header menu element) in the previous domain file's fragmentation point. Once the product is derived, (5) a copy of the *header.jsp* is included in the product folder (application file) and the *alterHeader.frag* injects (6) the new menu element over the derived application file.

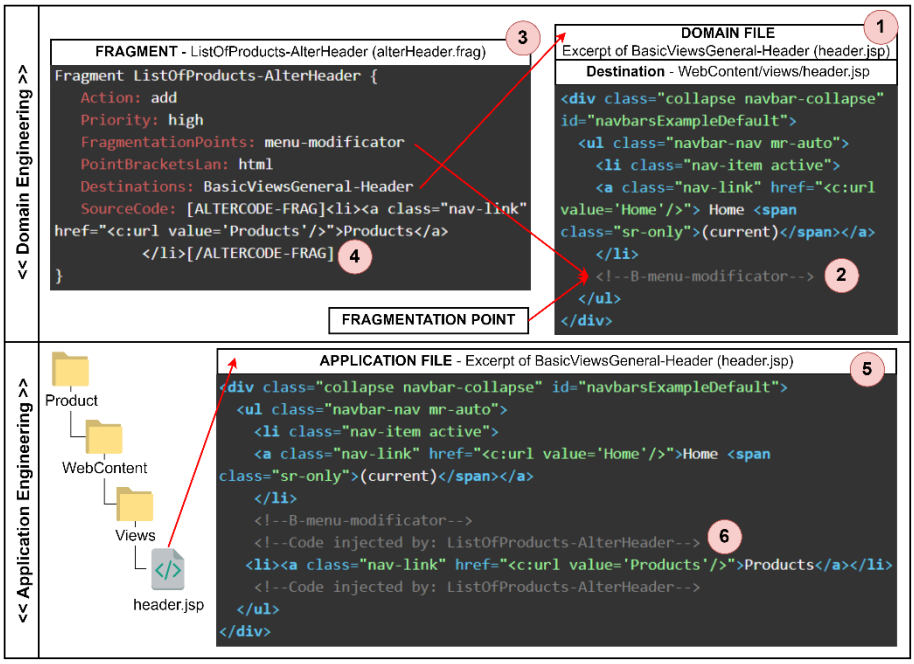

Fig. 1. An assembling scenario using VariaMos (FragOP).

2.2 VariaMos

VariaMos is a modeling tool and a framework that supports the FragOP approach [6]. VariaMos allows (i) specifying the PL requirements in the form of a "Feature model", (ii) specifying the PL domain components, (iii) linking PL requirements with the domain components that implemented them, (iv) configuring products, (v) deriving products, (vi) customizing products, and (vii) verifying products. One of the last stable versions of VariaMos was designed as a web application with Vue.js and some optional back-end servers [7].

We selected to use and enhance FragOP and VariaMos to support the definition of SPL composed of interconnected applications and to automate the deployment of the generated applications to the Cloud. It is due to the FragOP and VariaMos characteristics: (i) FragOP supports the construction and assembling of assets implemented over different software languages (which is key for interconnected applications developed with different software languages). (ii) FragOP supports the customization of assets (which is key to defining specific environmental variables of interconnected applications). (iii) VariaMos is an open-source project, and we have much experience working with this tool. Moreover, (iv) VariaMos is a web application that facilitates connecting with cloud services such as GitHub or the Google Cloud Platform (GCP).

In the next section, we describe a running example that will be used to exemplify the enhancement of FragOP and VariaMos.

3 Running Example

Running examples have been used to illustrate the concepts and describe the process of an investigation [8]. In this case, we will use a running example to provide a practical understanding of the new proposal and capabilities.

Our running example is a To-Do SPL. The To-Do SPL will permit to derive products that allow: (i) to create tasks, (ii) to delete tasks, (iii) to upload images for specific tasks, (iv) to sort tasks by their name, and (v) to check if a task has been completed. Primarily, the derived To-Do products allow users to manage their custom tasks. This To-Do SPL could be designed as a single self-contained application. However, we increased this project's complexity by creating a To-Do SPL composed of two interconnected applications (a front-end and a back-end), allowing us to exemplify the new approach enhancements.

The main idea with this complexity is to derive and deploy different front-end products, which will be interconnected with the derivation and deployment of different back-end products. To represent this complexity, we designed the To-Do feature model (see Fig. 2). In this feature model, we grouped the front-end and back-end requirements under respective features. This model will allow us to derive different front-end and back-end applications. The Frontend feature is optional, so it is possible to derive applications that contain only the Backend (i.e., if we only need a back-end product). We will explain in detail this complete process in the next section.

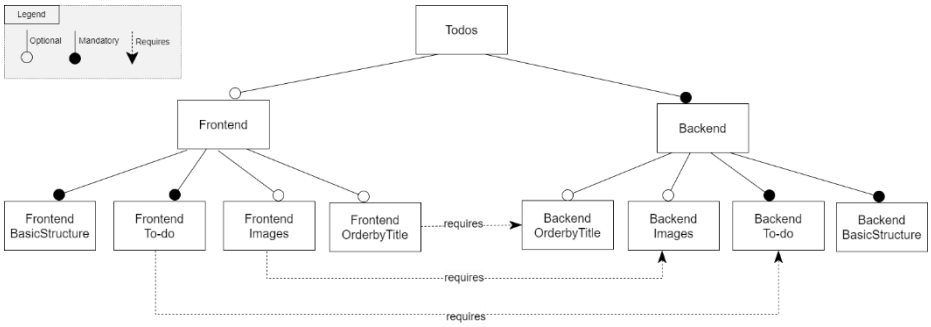

Fig. 2. To-Do SPL feature model.

Figure 3.A shows a derived and deployed front-end product which includes the selection of the "Frontend images" and "Frontend OrderByTitle" requirements (that internally contains a complete back-end product). On the other hand, Figure 3.B shows a derived and deployed front-end product that does not include the "Frontend images" and "Frontend OrderByTitle" requirements (that internally contains a back-end product without those elements). Therefore, we highlighted in red the elements that contain the first derived front-end product that does not contain the second derived front-end product.

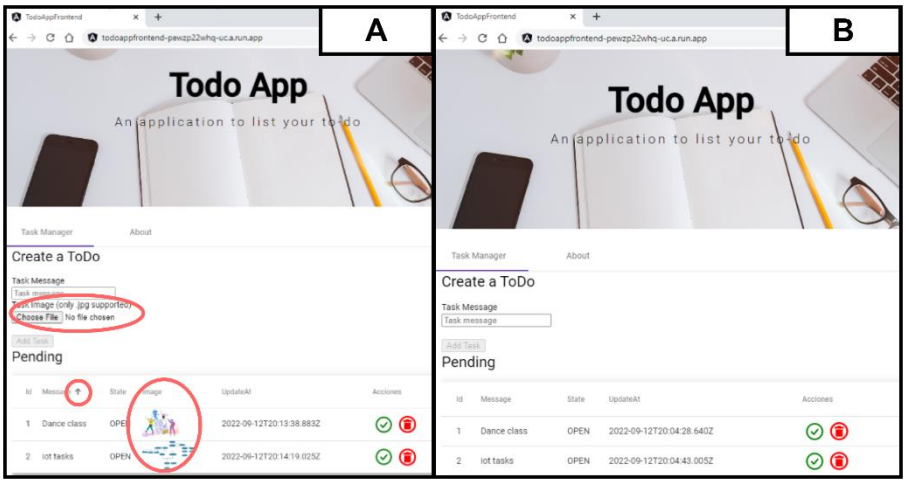

Fig. 3. Two different derived To-Do applications running on the Cloud.

Figure 4 shows the architecture of one of the previously derived To-Do applications. It shows that clients connect to the application through the HTTP protocol (to the front-end), and the front-end requests information through the REST mechanism to the back-end. The back-end collects the information from an SQLite database and returns it in a JSON format, later presented to the clients in their browsers. Besides, it shows that both front-end and back-end are deployed in different GCP instances.

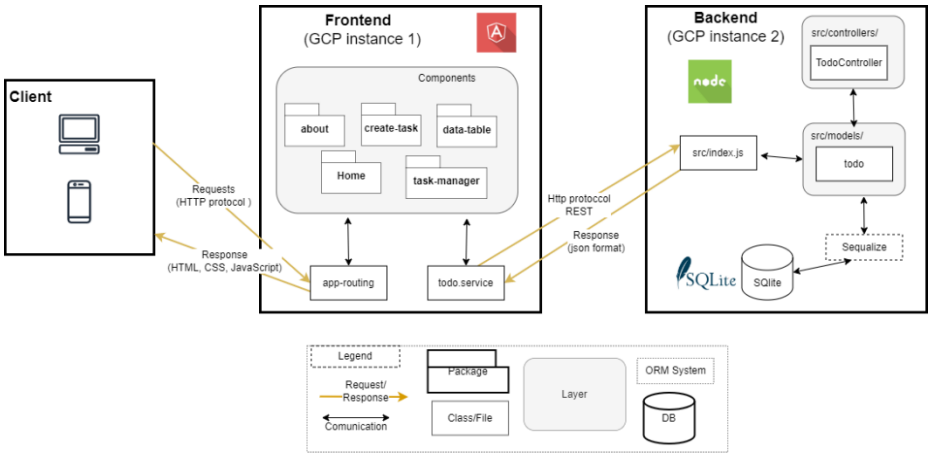

Fig. 4. A To-Do derived product architecture.

To implement and deploy applications, as shown in Figures 3 and 4, we need to create reusable components for the front-end application (that is developed in Angular), reusable components for the back-end application (that is developed in node.js and express), and a mechanism to interconnect those applications. The next section will explain how these components were created, how the To-Do SPL was designed, and how these products were derived and deployed.

4 Extending FragOP and VariaMos to support SPL composed of interconnected applications and automate products deployment

To implement an SPL under the FragOP approach, a developer should follow the FragOP process. The FragOP process consisted of eight activities explained in previous work [6]. We created a new SPL implementation process based on the FragOP process. The new process supports the definition of SPL composed of interconnected applications and automates the deployment of the generated applications to the Cloud.

This section explains the ten activities of the new SPL implementation process (cf. Fig. 5). Activities marked in bold were slightly modified (from the previous FragOP process work), and the activities with a grey background were added to support the deployment functionality.

This process is divided into (i) **domain engineering** which consists of designing and implementing the SPL reusable domain assets, and (ii) **application engineering** which consists of generating specific products for specific customers by reusing the domain assets.

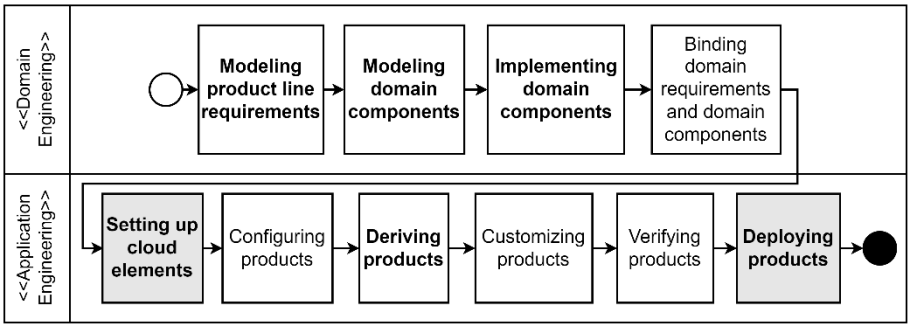

Fig. 5. New SPL implementation process (UML activity diagram).

4.1 Modeling product line requirements

The first activity consists of representing the domain requirements and their variability. Our approach allows specifying the SPL requirements as a "Feature model". However, since we want to represent an SPL composed of interconnected applications, we suggest grouping the different applications' requirements around a common ancestor. Figure 2 shows the feature model of our To-Do SPL running example. Requirements of the front-end domain application and back-end domain application were separated based on different ancestors (features). Besides, the front-end ancestor is optional, which means that this SPL can generate, and derivate To-Do products only composed of a back-end application. Generation of only back-end applications is common in projects divided into a back-end and a front-end (such as a flight or a hotel reservation system).

4.2 Modeling domain components

The second activity involves designing a component model representing the SPL domain components, their domain files, and the relationship between these elements. In the previous work, all domain components had the same hierarchy. However, since we now have interconnected applications, we added a new component model element called "App". This element allows us to group components that belong to different applications. For example, Figure 6 shows an excerpt of the component model of our To-Do SPL running example. As shown, we have two app elements (back-end and front-end) at the top of the hierarchy, then the corresponding components are linked to those apps.

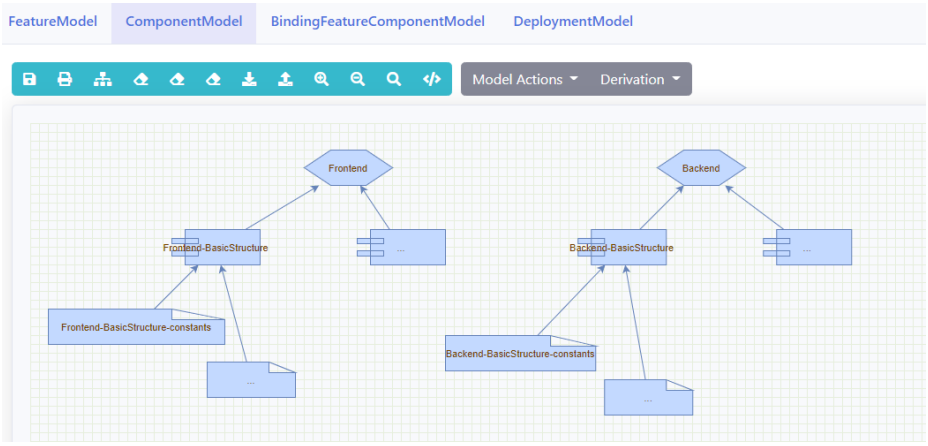

Fig. 6. An excerpt of the To-Do SPL component model in VariaMos.

4.3 Implementing domain components

This activity involves developing the applications, domain components, and files (based on the previous component model). This activity implies using FragOP to develop the component files code, the fragmentation points, the fragments, the customization points, and the customization files. This process remains similar to the previous work [6]. The only difference is that components are now grouped inside an application folder to differentiate the components that belong to the different applications. Figure 7 shows the two application folders (back-end and front-end) with their respective components.

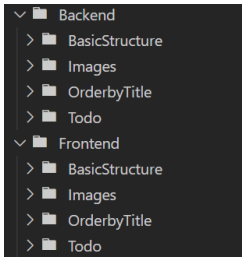

Fig. 7. To-Do SPL application folders and their respective components.

For this specific To-Do SPL, we included two configuration files (called *cloudbuild.json*) inside the *Frontend-BasicStructure* and the *Backend-BasicStructure* components. These files contain a template with instructions to deploy the derived products into the Cloud (in this case, inside a Google Cloud virtual machine). These templates are later customized (in Section 4.8) with the real google cloud builders' information that the SPL customers want to use. The complete code of the To-Do SPL domain components can be found in this GitHub repo [9].

4.4 Binding domain requirements and domain components

This activity consists of linking the components and requirements. It allows specifying what domain requirements (features) are implemented by what domain components. The process remains the same as the previous FragOP version. Figure 8 shows the binding model of our To-Do SPL running example.

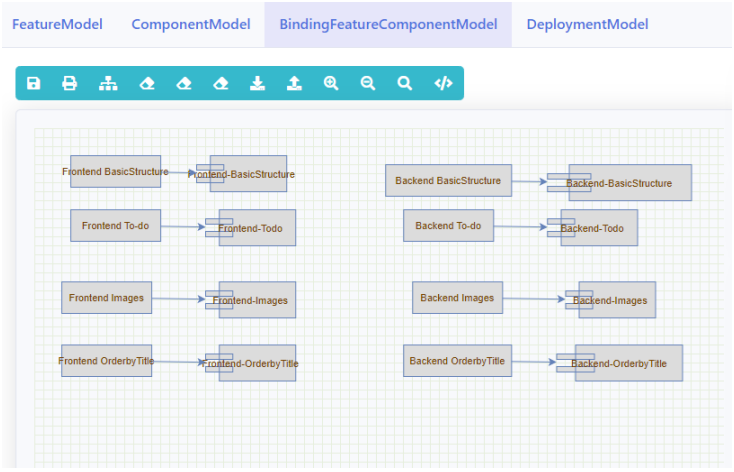

Fig. 8. To-Do SPL binding model in VariaMos.

This activity completes the SPL domain engineering process. Now, we will see the SPL application engineering process, which consists of generating specific products for specific customers.

4.5 Setting up cloud elements

Setting up cloud elements is the first activity of the application engineering process. This activity will allow us to automate the SPL products' deployment to the Cloud. We designed a complete step-by-step tutorial to be able to derivate and deploy To-Do projects to the Cloud [10]. In this activity, we need to configure a set of elements to upload the product code to platforms such as GitHub and to deploy the code to Cloud services such as Google Cloud Platform. This activity includes the next steps: (i) Create GitHub repositories, (ii) Create and configure a GitHub token, (iii) Create a Google Cloud Platform (GCP) project, and (iv) Create Google Cloud Build Triggers.

4.5.1 Create GitHub repositories

The customer or the SPL team must create GitHub repositories based on the amount of SPL interconnected applications which want to derive and deploy on the Cloud. Based on our running example, if a customer wants to configure a To-Do product that includes an interconnected front-end and back-end applications, then the customer or the SPL team must create two empty GitHub repositories (one for the back-end appli-

cation code, and another for the front-end application code). These new repositories should have at least one file (in the *main* branch), such as a README file, to work with VariaMos. In our case, we created two GitHub repositories, one for the back-end code [11] and another for the front-end code [12].

4.5.2 Create and configure a GitHub token

The customer or the SPL team must create a GitHub token with complete access to the previously created repositories. This token will allow us to connect VariaMos with those repositories and apply changes inside those repositories (such as uploading the derived code). Then, a developer must include the previous token in the VariaMos config section.

4.5.3 Create a Google Cloud Platform (GCP) project

We designed the To-Do SPL running example to work with GCP. In this case, the customer or the SPL team must create a GCP project. However, even when the running example is designed to work with GCP, similar steps and processes can be performed to connect with Microsoft Azure, Amazon Web Services, or similar cloud services.

4.5.4 Create Google Cloud Build Triggers

GCP offers a service called Cloud Build. Cloud Build is a service that allows us to build, test and deploy containers continuously. For example, we designed both back-end and front-end applications as containers (with Docker). So, we will be able to easily deploy these applications with a service such as Cloud Build.

Cloud Build also offers a service called triggers. A Cloud Build trigger automatically starts a build whenever we make changes to our application source code. Since we wanted to derive two products (back-end and front-end), we created two Cloud Build triggers (see Figure 8), connected to the back-end and front-end GitHub repositories. At the beginning, the repositories are empty, but once we apply some changes to those repositories, the triggers will be executed, and the applications will be deployed to the Cloud.

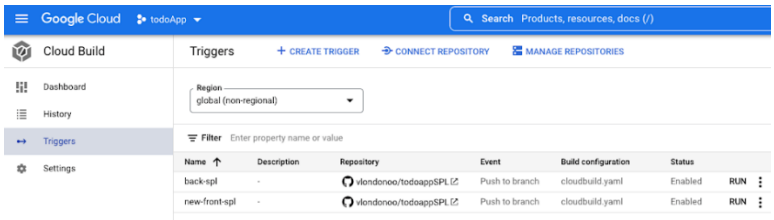

Fig. 8. Back-end and front-end Cloud Build triggers in GCP.

4.6 Configuring products

Configuring products involves selecting the specific features that a specific product will contain based on the stakeholder requirements. For this example, we selected all the leaf features in VariaMos (see Figure 9). It means we want to generate both back-end and front-end applications with all the available characteristics.

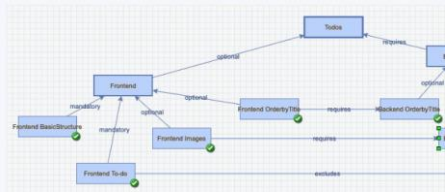

Fig. 9. To-Do SPL product configuration with all leaf features selected in VariaMos.

4.7 Deriving products

Deriving products consists in generating specific software products based on the SPL configuration. VariaMos uses and assembles reusable domain assets to generate new products. The process consists of: (i) the selected leaf features are taken as an input, (ii) the binding is resolved to show what components should be assembled based on the selected features, and (iii) the components are assembled over different products folders depending on the app they belong (the output). Therefore, VariaMos executes the fragments which modify the product application file code. Figure 10 shows how to execute the product derivation in VariaMos.

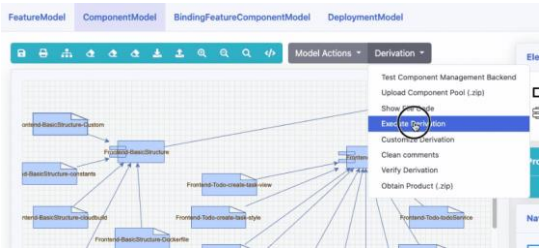

Fig. 10. To-Do SPL product derivation in VariaMos.

4.8 Customizing products

Customizing products involves modifying the derived products based on the customer's credentials or needs. In our case, the front-end product contains a file with a dummy URL to the back-end product. We need to customize this file to put the real back-end URL (this specific front-end customization can only be applied once we deploy the back-end product). Therefore, both the front-end and back-end contain a *cloudbuild.json* file which needs to be customized with the custom GCP project name and other custom variables.

Additionally, we added a new option called “Clean derivation” in VariaMos. This option removes fragmentation and customization points (annotations) over specific derived files. For example, we have an annotation over a *package.json* file that needs to be removed to be able to execute that file.

The complete customization and clean process can be found here [10].

4.9 Verifying products

VariaMos supports a simple product verification process. Based on the derived files extension, VariaMos analyses the grammar and syntax of each derived file and generates alerts if errors are found.

4.10 Deploying products

The last activity consists of deploying the products to the Cloud. First, VariaMos creates a copy of each “app” defined in the “Component Model”, into a new “Deployment Model”. This model only contains a new option called “Upload Apps to GitHub”. In this option, we select the app we want to upload to GitHub and put the GitHub repository name. Figure 11 shows that we want to push our back-end derived code to one of our GitHub repositories.

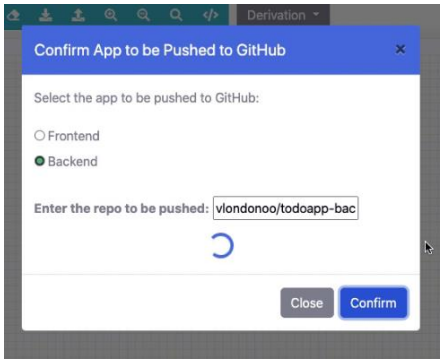

Fig. 11. Pushing the back-end derived code to GitHub.

After a few minutes, the Google Cloud Build back-end trigger automatically starts, builds the project, and deploys the back-end product to the Cloud. Next, it is deployed into a service called Cloud Run, which is a serverless service. Then, we have the back-end product running over a custom Cloud Run URL on the Internet. We can now take that custom URL, customize the front-end with that URL and repeat the process to deploy our front-end product. Figure 12 shows the derived back-end and front-end applications running on Cloud Run.

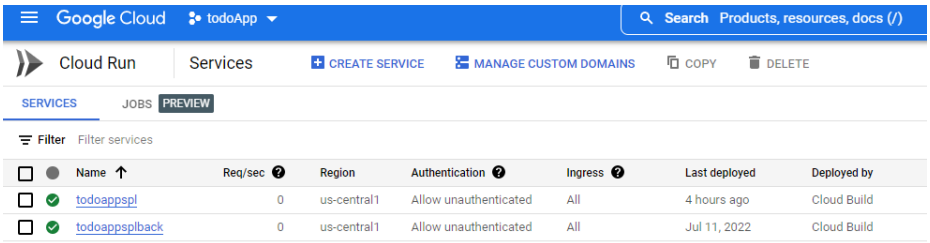

Fig. 12. Derived back-end and front-end applications running on Cloud Run.

5 Deployment results

To evaluate the new approach, we derived and deployed five different products based on our To-Do SPL running example. We also capture some important data of the derivation and deployment process. Next, we describe the five products:

- **P1:** it was configured by selecting all To-Do SPL Backend available features.
- **P2:** it was configured by selecting all To-Do SPL Backend and Frontend mandatory features ("Frontend BasicStructure", "Frontend To-do", "Backend BasicStructure", "Backend To-do").
- **P3:** it was configured by selecting all To-Do SPL Backend and Frontend mandatory features, plus "Frontend OrderByTitle" and "Backend OrderByTitle".
- **P4:** it was configured by selecting all To-Do SPL Backend and Frontend mandatory features, plus "Frontend Images" and "Backend Images".
- **P5:** it was configured by selecting all To-Do SPL Backend and Frontend features.

Figure 13 shows the lines of code (LOC) reused, injected, and customized in the derivation of each of the previous products (for both back-end and front-end applications). It shows that the back-ends were developed mainly using JavaScript (JS) and JSON, and the front-ends were developed mainly using JavaScript (JS), TypeScript (TS), JSON and HTML. It also shows that most of the derivation process was automated. Only ten LOC were manually customized per product derivation.

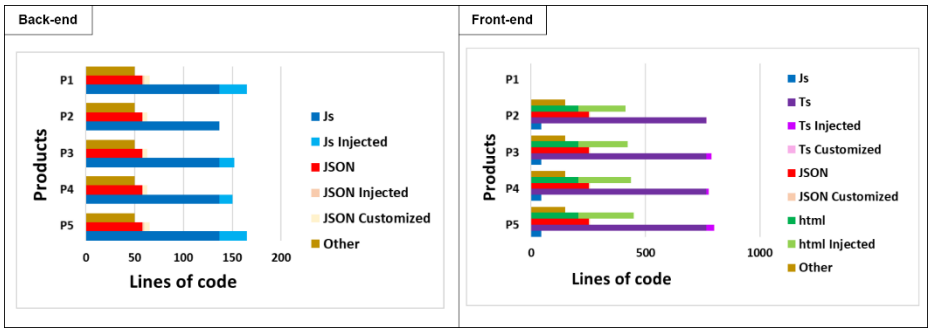

Fig. 13. LOC reused, injected, and customized in the derivation of each product.

Table 1 shows the product deployment results. It shows each product with its corresponding: (i) type of derived applications, (ii) selected leaf features (during the configuration process), (iii) linked component files (to the corresponding selected leaf features), and (iv) time to deploy to the Internet.

Table 1. Product deployment results.

Product	Derived applications	Leaf features selected	Linked component files	Time to deploy (seconds)
P1	Back-end	5	18	65
P2	Back-end and front-end	6	27	144
P3	Back-end and front-end	8	32	155
P4	Back-end and front-end	8	34	166
P5	Back-end and front-end	10	40	166

In average, it took 2 minutes and 37 seconds to deploy products that contained both back-end and front-end applications (P2 to P5). The time was calculated from when the SPL team clicked the "Upload apps to GitHub" in VariaMos, until the application was ready to use on the Internet. All this deployment process was automated by connecting Google Cloud Build Triggers, and Google Cloud Run with the GitHub repositories.

6 Related work

During the last years, different authors have tried to integrate the SPL component development and the product derivation with different DevOps practices, to increase efficiency, speed, security, and delivery of the SPL projects. Casquina & Montecchi [3] proposed an approach to integrate the conditional compilation mechanism used to implement the SPL variabilities and the Git version control system used to manage software versions to increase the attractiveness of the SPLs in the industry.

Authors have also proposed new ways of designing and implementing more complex SPL applications. Tizzei *et al.* [13] investigated the integrated use of microservices architecture and software product line techniques to develop multi-tenant SaaS. They developed an empirical study that showed an average software reuse of 62% of lines of code among tenants. Trujillo-Tzanahua *et al.* [14] presents a study about the application of Multi Product Lines (MSPL) in the software development process. A MSPL is a software product line that results from combining components or products developed from several independent and heterogeneous software products lines [15]. They showed the importance and usefulness of applying MSPL approaches and discussed some challenges. Benni *et al.* [16] analyzed service dependencies as feature dependencies, at the feature, structural, technological, and versioning level, to assess the interchangeability of services. They analyzed six community-selected use-cases and reported that services are non-interchangeable systematically.

7 Conclusions

The SPL community needs to provide more robust methodologies and tools to support many common software industry necessities. This paper presented an enhanced version of FragOP and VariaMos to support the definition of SPL composed of interconnected applications and automate the deployment of the generated applications to the Cloud. With the enhanced approach, we covered some important industry necessities. Now, SPL developers can design, implement, derive, and deploy complex software products to the Cloud. And many of these steps were automated. We also developed a running example (a ToDo SPL) to show some preliminary results of the enhanced approach. We derived five different products, four of them consisted in an interconnected back-end and front-end applications, and we deployed all of them to the Cloud (by using GitHub and the GCP platform). As a future work, we plan to study how to integrate testing activities, how to elaborate more complex SPL projects, and design and implement more rigorous experiments.

References

1. P. Clements. Software product lines: Practices and patterns. Boston: Addison-Wesley, 2002.
2. T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake and T. Leich. FeatureIDE: An extensible framework for feature-oriented software development, Science of Computer Programming, vol. 79, pp. 70–85, 2014.
3. J. C. Casquina and L. Montecchi. A proposal for organizing source code variability in the git version control system, in International Systems and Software Product Line Conference, Leicester United Kingdom. New York, NY, USA: ACM, 2021.
4. R. Mazo, J.C. Muñoz-Fernández, L. Rincón, C. Salinesi, and G. Tamura, VariaMos: an extensible tool for engineering (dynamic) product lines. In: SPLC, pp. 374–379, 2015.

5. Montalvillo, O. Díaz and M. Azanza. Visualizing product customization efforts for spotting SPL reuse opportunities, in International Systems and Software Product Line Conference, Sevilla Spain. New York, NY, USA: ACM, 2017.
6. D. Correa, R. Mazo and G. L. Giraldo. Extending FragOP Domain Reusable Components to Support Product Customization in the Context of Software Product Lines, in Lecture Notes in Computer Science. Cham: Springer International Publishing, pp. 17–33, 2019.
7. VariaMos web, GitHub repository, Available at: https://github.com/VariaMosORG/VariaMos. Accessed: 04-12-2022.
8. J. C. Wileden and A. Kaplan. Software interoperability, in the 21st international conference, Los Angeles, California, United States, 1999.
9. Todo-app-variamos. GitHub repository, Available at: https://github.com/vlondonoo/todo-app-VariaMos. Accessed: 04-12-2022.
10. Todo-app-variamos wiki. GitHub repository, Available at: https://github.com/vlondonoo/todo-app-VariaMos/wiki. Accessed: 04-12-2022.
11. Todoapp-backend GitHub repository, Available at: https://github.com/vlondonoo/todoapp-backend. Accessed: 04-12-2022.
12. Todoapp-frontend, GitHub repository, Available at: https://github.com/vlondonoo/todoapp-frontend. Accessed: 04-12-2022.
13. L. P. Tizzei, M. Nery, V. C. V. B. Segura and R. F. G. Cerqueira. Using Microservices and Software Product Line Engineering to Support Reuse of Evolving Multi-tenant SaaS, in International Systems and Software Product Line Conference, Sevilla Spain. New York, NY, USA: ACM, 2017.
14. G. I. Trujillo-Tzanahua, U. Juárez-Martínez, A. A. Aguilar-Lasserre and M. K. Cortés-Verdín. Multiple Software Product Lines: applications and challenges, in Advances in Intelligent Systems and Computing. Cham: Springer International Publishing, pp. 117–126, 2017.
15. R. Rabiser, P. Grünbacher and M. Lehofer. A qualitative study on user guidance capabilities in product configuration tools, in International Conference, Essen, Germany. New York, New York, USA: ACM Press, 2012.
16. B. Benni, S. Mosser, J.-P. Caissy and Y.-G. Guéhéneuc. Can microservice-based online-retailers be used as an SPL?, in International Systems and Software Product Line Conference, Montreal Quebec Canada. New York, NY, USA: ACM, 2020.