Extending a low-code tool with multi-cloud deployment capabilities

Fitash Ul Haq[1*], Iván Alfonso[1*†], Armen Sulejmani[1], and Jordi Cabot[1,2]

[1] Luxembourg Institute of Science and Technology, Esch-sur-Alzette, Luxembourg
{fitash.ulhaq,ivan.alfonso,armen.sulejmani,jordi.cabot}@list.lu
[2] University of Luxembourg, Esch-sur-Alzette, Luxembourg jordi.cabot@uni.lu

Abstract. Low-code emerged as an evolution of model-driven engineering to accelerate software delivery, and it continues to gain traction today. However, low-code tools and solutions have primarily focused on development, often neglecting or offering minimal support for the application deployment process, such as lacking capabilities for multi-cloud deployments. In this paper, we propose an extension of BESSER, an open-source low-code platform, to address the packaging and deployment of applications in multi-cloud environments. This extension includes the definition of a language and a grammar to enable the modeling of the deployment architecture, also enabling the specification of public and on-premises clusters. Additionally, we have developed code generators to automate the application packaging, and cloud provisioning and deployment using Terraform. The complete infrastructure is available in an open-source repository.

Keywords: low-code · deployment architecture · multi-cloud.

Demo video: http://tiny.cc/demo-video

1 Introduction

In recent years, low-code tools have been expanding both in academia and software industry. These tools enable developers to focus more effort on business-related tasks as development complexity is reduced. Low-code tools aim to reduce the amount of manual-coding required to accelerate software delivery by raising the level of abstraction to facilitate domain specification and ignore irrelevant technical details. It can be seen as an evolution or continuation of model-based approaches [4], whose foundations are based on model-driven engineering (MDE). However, low-code should not only focus on accelerating application development, but also on supporting different processes throughout the software lifecycle such as testing, deployment, monitoring, and adaptation.

Usually, low-code tools omit, or provide little support for modeling other aspects such as deployment architecture (also known as hardware architecture[6]),

*These authors contributed equally.
†Corresponding author.

i.e., aspects related to infrastructure, clustering, virtualization technology, and app deployment. For example, the specification of multi-cloud deployments that use several cloud providers to deploy the different application components or hybrid deployments that include on-premises clusters, are usually not supported.

To address this concern, we have developed an extension of the BESSER low-code platform [2] to (1) enable the specification of the deployment architecture of the target software and (2) automate the packaging, provisioning and deployment of the system according to that deployment model. This extension aims to transform BESSER into a low-code platform that can also cover these final phases of the software development lifecycle.

The remainder of the paper is organized as follows: Section 2 presents an overview of the low-code tool extension. Details on modeling and code generation are introduced in Section 3 and 4 respectively. Related work is discussed in Section 5, and Section 6 concludes the paper.

2 Overview of Extended BESSER Platform

As part of this work, we have extended the BESSER platform (1) by equipping it with a complete backend generator capable of producing the schema of a database and a REST API providing an abstraction layer for CRUD operations that can be packaged as a docker container, and (2) by providing BESSER with the capability of modeling the software architecture and, based on that model, deploying applications on a cloud infrastructure, making it available for end users to use. To perform the latter, we have introduced a new metamodel and grammar to parse the deployment model in textual form (details in section 3), and a code generator (see section 4) to generate an Infrastructure as Code definition that can be directly run to provision and deploy applications on the cloud.

This architectural view is combined with the more "traditional" models defining the structural and behavioural aspects of the application. Indeed, at the core of BESSER we have B-UML (BESSER's Universal Modeling Language), a foundational language designed for specifying and modeling various aspects of a system. This language supports the creation of different types of models, including structural or data models, object models, graphical user interface models, and even specifying OCL constraints. B-UML models serve as the input for code generators that produce application code. BESSER provides code generators for various technologies such as SQLAlchemy, Django, and Python.

Figure 1 shows the overview of extended BESSER platform combining the two perspectives. Both types of models are defined using a textual notation that conforms to their respective grammars. The BESSER platform parses these models and creates their respective B-UML models. Code generators interpret these B-UML models to produce various software artifacts, including the scripts for configuring and deploying the application on the multi-cloud environment, which is accessible by users.

Internally, BESSER first parses the structural model and generates a B-UML structural model that conforms to B-UML metamodel. This B-UML metamodel

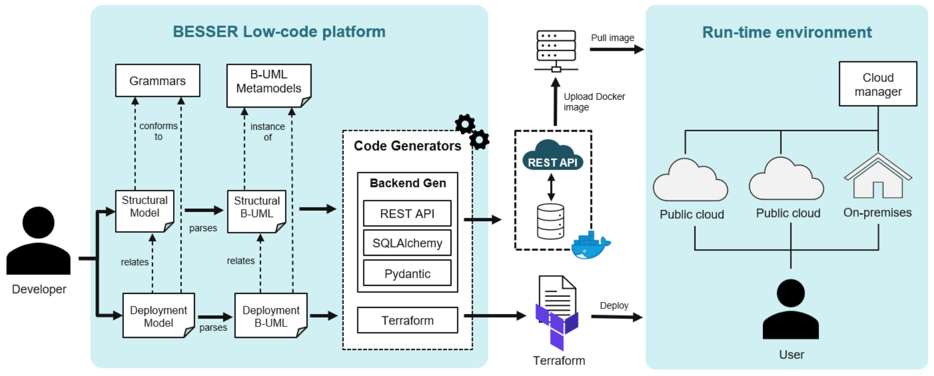


Fig. 1. Overview of extended BESSER platform

is then used by code generators to produce the core code for the application along with the database support. In a second step, the platform generates python scripts to containerize the application along with the database. BESSER uses this script to upload the image to the hub automatically. BESSER currently supports DockerHub for this purpose but can be easily extended for other repositories. In a third step, the platform parses the deployment model and generates a deployment B-UML model, that conforms to B-UML deployment architecture metamodel, which is then used by code generators to generate the Terraform[3] scripts. Terraform is a state-of-the-art open source tool developed to deploy applications on the cloud infrastructure. Finally, these Terraform scripts are run to configure the cloud infrastructure (including nodes, clusters, networks, etc.) and deploy the containerized application using the image from the hub (created in step 2). Additionally, a load balancer is also provisioned to distribute the web traffic between the nodes in the clusters.

3 A Modeling Extension for Specifying a Deployment Architecture

To deploy application on the cloud, BESSER platform requires two models from the user: (1) a structural model that contains information about the application itself and (2) a Deployment model that contains information about the target cloud-based deployment architecture.

Due to the space limitations, we do not discuss the structural model in this paper. More details on this model can be found in this previous work [2]. Therefore, the rest of this section focuses on the deployment model.

Figure 2 shows the metamodel we have introduced for modeling deployment architecture of cloud infrastructure. The metamodel contains a root *DeploymentModel* metaclass, which is the main construct of the model. It comprises all the information about the cloud infrastructure. A *DeploymentModel* can have one or

[3]https://www.terraform.io/

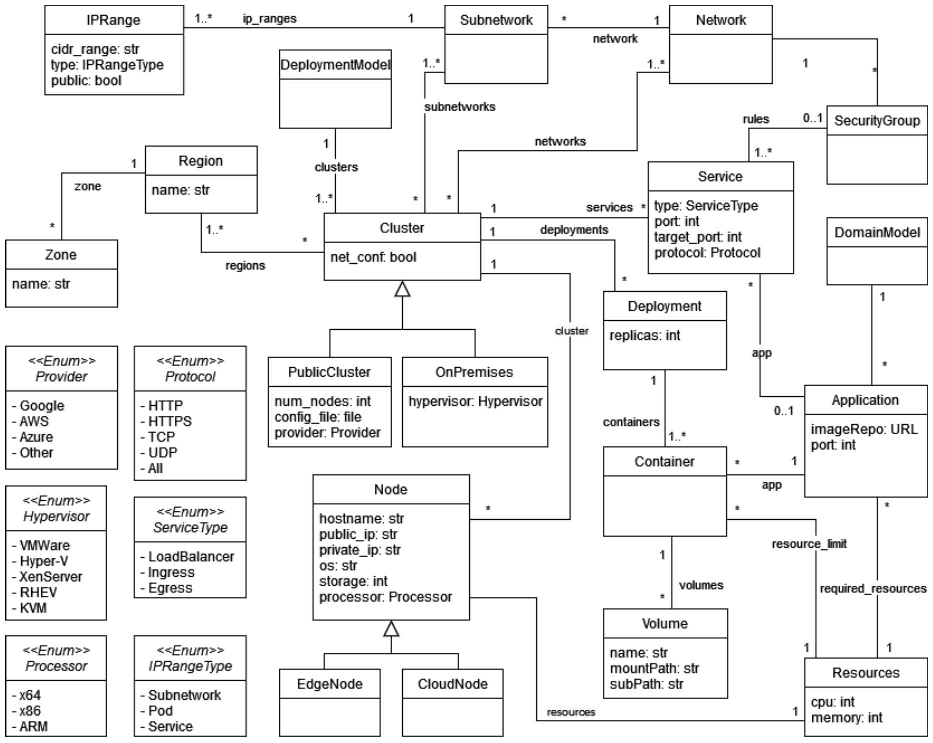


Fig. 2. Deployment Architecture Metamodel

multiple *Cluster*(s). Each *cluster* can be a *PublicCluster* or an *onPremise* one. For a *publicCluster*, the *config_file* contains the credentials and other information needed to communicate with the public cloud provider. Additionally, the architect needs to define five items for each cluster: (1) **Region**, (2) **Network**, (3) **Node**, (4) **Deployment**, and (5) **Services**. We discuss these items briefly:

As part of the region information, the architect needs to define the region/s and zones where the cluster should be created. For the network information, the architect needs to define networks, subnetworks, and IP ranges for these networks; in order to make it easier, we have a *net_conf* boolean variable to auto-generate the network and subnetworks. Regarding the node information, the OS, storage, resources, and processor for the *node*s of cluster must be defined. Each *node* can be of *CloudNode* or *EdgeNode*. As part of the fine-grained deployment information, the architect needs to specify the number of replicas of each *Deployment*; each *deployment* must have a container containing the dockerized application, volume and resource information. As for the services information, we need to provide all the services that will be attached to the clusters. Services can be load balancers, ingress or egress. Furthermore, we can also define the port mappings and security groups for the services.

This information can be entered thanks to the textual notation (i.e. concrete syntax) we have provided for the above metamodel. We have designed a grammar and a parser that converts this textual description into a B-UML model. The grammar is created using ANTLR [7], a widely used language recognition tool. We show an example of a deployment model using this grammar below but the complete grammar is available in the project repository [1].

For the purpose of exemplification throughout this study, we use the Digital Product Passport (DPP) domain, a European initiative supporting the circular economy [11]. DPP involves collecting comprehensive information on a product's composition, design, condition, and other aspects throughout its life cycle. This information can be shared with anyone interacting with the product, including manufacturers, buyers, and repair staff.

Listing 1.1 shows an excerpt of the deployment model for the DPP example, written using our defined concrete syntax. In this example, the *dpp_ app* application is defined (*lines 3-5*) and packaged in the Docker image "dpp/app:latest". This image is created using the structural model, which defines the DPP domain, and is automatically uploaded to DockerHub by the backend generator. The application runs in the *dpp_ container* (*lines 8-11*), with 2 replicas deployed as defined in *dpp_ deployment* (*lines 14-16*). Finally, two public clusters (for Google and AWS) are defined (*lines 19-28*) to deploy the application.

Listing 1.1. Deployment model

```
 1  Deployment model{
 2      applications {
 3          ->  name: dpp_app,
 4              image: "dpp/app:latest",
 5              // Other info omitted for brevity purposes
 6          }
 7      containers {
 8          ->  name: dpp_container,
 9              app_name: dpp_app,
10              cpu_limit: 500m,
11              memory_limit: 512Mi }
12      deployments {
13          ->  name: dpp_deployment,
14              replicas: 2,
15              containers: [dppcontainer] }
16      clusters {
17          ->  public_cluster
18              name: cluster_a,
19              provider: google,
20              deployments: [dpp_deployment],
21              // Other info omitted for brevity purposes
22          ->  public_cluster
23              name: cluster_b,
24              provider: aws,
25              deployments: [dpp_deployment],
26          }
27  // Other concepts omitted for brevity purposes ...
28  }
```

4 Code generation

The code generation for this BESSER extension primarily involves two generators implemented as M2T transformations using the Jinja[4] template-based engine: one for the backend and one for the Terraform code.

The Backend generator integrates multiple specialized generators from the BESSER suite (see Figure 1) to create a comprehensive backend. It enables the creation of dynamic and scalable API endpoints using the REST API Generator, which leverages the FastAPI framework. Efficient ORM transformations for database interactions are handled by the SQLAlchemy generator, while robust data validation is ensured by the Pydantic generator, promoting data integrity and backend security. Furthermore, this generator can also package the backend as a container image and upload it to a hub for easy deployment.

On the other hand, the Terraform generator creates infrastructure as code (IaC) for provisioning and deploying the application generated using the backend generator. This code encompasses infrastructure configuration such as clusters, networks, subnets, and load balancers, as well as the deployment of software containers using Kubernetes management services like EKS[5] and GKE[6].

Using the same example from the DPP domain (introduced in section 3); Figure 3 shows the generated code for deploying the multi-cloud environment using Terraform. A set of files containing the provisioning and deployment configurations is generated for each cloud provider. The deployment process is initiated by executing the *setup.bat* file. For a detailed walkthrough and application execution, you can refer to the demo video accompanying this tool paper.

All files, code, and steps to run this example can be consulted in the BESSER example repository[7].

5 Related work

The deployment of applications in the cloud is a well-researched topic. However, most studies do not fully integrate their solutions within a software development workflow. This is particularly true for Infrastructure as Code (IaC) tools, which, although powerful for configuring deployments, are not designed to support other phases of the application development lifecyle. Nonetheless, these tools can be incorporated within a development and deployment framework, as we propose with our low-code tool.

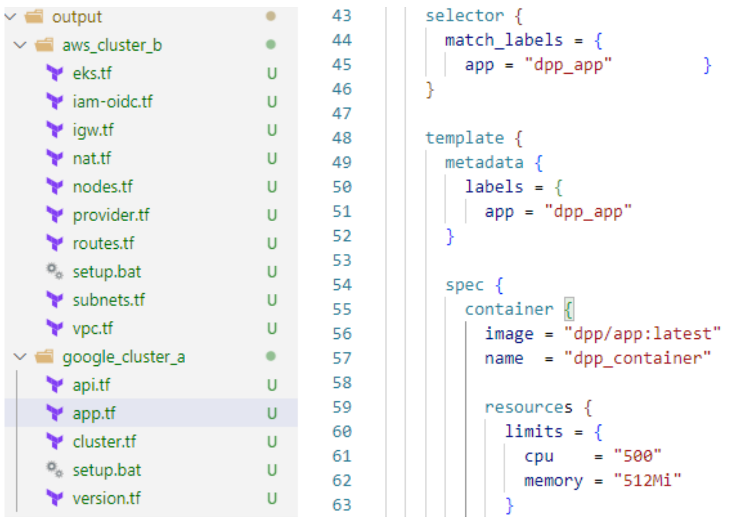
Some studies, such as [9, 10, 3], propose model-based solutions for specifying and automating cloud deployments. However, these studies do not address the modeling of multi-cloud and on-premises environments, nor do they incorporate containerization as a virtualization technology for application deployment. Other

[4]https://palletsprojects.com/p/jinja/

[5]https://aws.amazon.com/eks

[6]https://cloud.google.com/kubernetes-engine?hl=en

[7]https://github.com/BESSER-PEARL/BESSER-examples/tree/main/examples/multi-cloud_deploymen



```
    selector {
      match_labels = {
        app = "dpp_app"        }
    }

    template {
      metadata {
        labels = {
          app = "dpp_app"
      }

      spec {
        container {
          image = "dpp/app:latest"
          name  = "dpp_container"

          resources {
            limits = {
              cpu    = "500"
              memory = "512Mi"
            }
```

Fig. 3. Generated code for Terraform

works (such as [5, 8]) address the modeling and deploying architectures in multi-cloud environments but do not provide support for application development or containerized image support to facilitate deployment.

Commercial low-code tools also support cloud application deployment but have significant limitations in terms of configuration and customization. For instance, PowerApps and Appian typically restrict or suggest deployment to a single cloud provider, such as Azure and AWS, respectively. While some tools, like Mendix, OutSystems, and Pega, offer features for multi-cloud deployments, they limit infrastructure configuration and customization. Additionally, it is often necessary to install the low-code platform environment on a cloud provider's web server before deploying individual applications to that environment. Moreover, these tools are neither free nor open-source.

Despite the capabilities of existing solutions, there remains a gap in integrating low-code software development and multi-cloud deployment within a single tool. Our open-source low-code tool aims to bridge this gap by offering a unified framework for application development, Docker image generation, and deployment across multi-cloud environments.

6 Conclusions and Further Work

This paper extends the low-code tool, BESSER, with application packaging and multi-cloud deployment capabilities. This extension comprises several key components, including a new grammar and metamodel to specify the deployment architecture of the system, the generation of a Docker image for applications created with BESSER, and a new code generator capable of producing Terraform code for deployment across multiple cloud environments.

As part of our future roadmap, we aim to enhance the tool's usability by incorporating a new graphical notation, enabling graphical modeling of deployment architectures (e.g., using AWS Architecture Icons[8]). Additionally, we plan to enhance the capabilities of the Terraform code generator to support a wider range of cloud providers and on-premises clusters. We also plan to integrate a cloud management solution such as Google Anthos, enabling centralized management and monitoring of all clusters, including those deployed on-premises. Finally, we plan to address the continuous updating of models in runtime, enabling the integration of our tool into a continuous deployment pipeline.

Acknowledgments. This project is supported by the Luxembourg National Research Fund (FNR) PEARL program, grant agreement 16544475.

References

1. Github repo: BESSER. https://github.com/BESSER-PEARL/BESSER (2023)
2. Alfonso, I., Conrardy, A., Sulejmani, A., Nirumand, A., Ul Haq, F., Gomez-Vazquez, M., Sottet, J.S., Cabot, J.: Building besser: an open-source low-code platform. In: International Conference on Business Process Modeling, Development and Support. pp. 203–212. Springer (2024)
3. Artac, M., Borovšak, T., Di Nitto, E., Guerriero, M., Perez-Palacin, D., Tamburri, D.A.: Infrastructure-as-code for data-intensive architectures: a model-driven development approach. In: 2018 IEEE Int. Conf on Software Architecture (ICSA). pp. 156–15609. IEEE (2018)
4. Cabot, J.: Positioning of the low-code movement within the field of model-driven engineering. In: Proceedings of the 23rd ACM/IEEE Int. Conf. on Model Driven Engineering Languages and Systems: Companion Proceedings. pp. 1–3 (2020)
5. Ferry, N., Chauvel, F., Song, H., Rossini, A., Lushpenko, M., Solberg, A.: Cloudmf: Model-driven management of multi-cloud applications. ACM Transactions on Internet Technology (TOIT) **18**(2), 1–24 (2018)
6. Muccini, H., Sharaf, M.: Caps: Architecture description of situational aware cyber physical systems. In: 2017 IEEE Int. Conf. on Software Architecture (ICSA). pp. 211–220. IEEE (2017)
7. Parr, T.: The definitive antlr 4 reference. The Definitive ANTLR 4 Reference pp. 1–326 (2013)
8. Pham, L.M., Tchana, A., Donsez, D., Zurczak, V., Gibello, P.Y., De Palma, N.: An adaptable framework to deploy complex applications onto multi-cloud platforms. In: The 2015 IEEE RIVF Int. Conf. on Computing & Communication Technologies-Research, Innovation, and Vision for Future (RIVF). pp. 169–174. IEEE (2015)
9. Sandobalin, J., Insfran, E., Abrahão, S.: Argon: A model-driven infrastructure provisioning tool. In: 2019 ACM/IEEE 22nd Int. Conf. on Model Driven Engineering Languages and Systems Companion (MODELS-C). pp. 738–742. IEEE (2019)
10. Sledziewski, K., Bordbar, B., Anane, R.: A dsl-based approach to software development and deployment on cloud. In: 2010 24th IEEE Int. Conf. on Advanced Information Networking and Applications. pp. 414–421. IEEE (2010)
11. Walden, J., Steinbrecher, A., Marinkovic, M.: Digital product passports as enabler of the circular economy. Chemie Ingenieur Technik **93**(11), 1717–1727 (2021)

[8] https://aws.amazon.com/architecture/icons/