DevOps Round-trip Engineering: Traceability from Dev to Ops and Back Again

Miguel Jiménez[1], Lorena Castaneda[1], Norha M. Villegas[2], Gabriel Tamura[2], Hausi A. Müller[1], and Joe Wigglesworth[3]

[1] University of Victoria, Victoria, British Columbia, Canada
{miguel,lcastane,hausi}@uvic.ca
[2] Universidad Icesi, Cali, Valle del Cauca, Colombia
{nvillega,gtamura}@icesi.edu.co
[3] IBM Toronto Laboratory, Toronto, Canada
wiggles@ca.ibm.com

Abstract. DevOps engineers follow an iterative and incremental process to develop Deployment and Configuration (D&C) specifications. Such a process likely involves manual bug discovery, inspection, and modifications to the running environment. Failing to update the specifications appropriately leads to technical debt, including configuration drift, *snowflake* configurations, and erosion across environments. Despite the efforts that DevOps teams put into automating operations work, there is a lack of tools to support the development and maintenance of D&C specifications. In this paper, we propose TORNADO, a two-way Continuous Integration (CI) framework (*i.e.*, Dev $\xrightarrow{\text{CI}}$ Ops and Dev $\xleftarrow{\text{CI}}$ Ops) that automatically updates D&C specifications when the corresponding system changes, enabling bi-directional traceability of the modifications. TORNADO extends the concept of CI, integrating operations work into development by committing code corresponding to manual modifications. We evaluated TORNADO by implementing a proof of concept using Terraform templates, OpenStack and CircleCI, demonstrating its feasibility and soundness.

Keywords: DevOps, Round-Trip Engineering, Traceability, Software Deployment, Continuous Integration

1 Introduction

Changes in the artefacts used throughout software development and operations are inherently causally connected to one another. For example, modifying the deployment specifications will affect the corresponding system and the infrastructure it runs on. Analogously, updating the physical infrastructure will cause updates to the software and networking configuration. Traditionally, this relationship has been implicit and poorly supported by software development processes and tools. DevOps practices have increased its visibility in the context

of a continuous development process [1,2], impacting mostly the forward direction (*i.e.*, Dev $\rightarrow$ Ops). In contrast, there is a lack of standard and technology-supported processes to bridge explicitly and repeatedly in the backward direction (*i.e.*, Dev $\leftarrow$ Ops) [3,4,5,6]. This inability hinders the process of keeping operation and development information consistent with the deployed system.

Many organisations adopt a forward-only development strategy to avoid configuration inconsistency. Any modification to the system or its infrastructure must be performed in the forward direction, using, for example, infrastructure as code (IaC). This approach ensures consistency between the running system and its D&C specifications, and at the same time allows tracing the changes. However, DevOps engineers and operators still follow a manual bug discovery and exploratory experimentation process that leads to fixing faults. D&C specifications are the result of an incremental process, in which each step is likely to involve manual actions and inspection. Therefore, it is still a task of the engineers to capture the drift between an experimental environment and the original setting, before updating the specifications. Failing to do so leads to configuration drift, *snowflake* configurations, erosion across environments, and other forms of technical debt [7,8,9]. There is a need to support keeping the D&C specifications in sync.

Automatically maintaining the consistency between D&C specifications and a running system is known as automatic Round-Trip Engineering (RTE) [10,11,12]. Our contributions are as follows. We introduce Tornado, a framework for realizing RTE in DevOps. We demonstrate how the concept of continuous integration [13] can be extended from its traditional use to integrate operations work into development. Tornado is a two-way con**T**inu**O**us integ**R**atio**N** fr**A**mework for **D**ev**O**ps (*i.e.*, Dev $\xrightarrow{\text{CI}}$ Ops and Dev $\xleftarrow{\text{CI}}$ Ops) that enables bidirectional traceability [14] of changes and transformation between a running system and its D&C specifications. Our evaluation consists of a proof of concept implementation based on Terraform templates[4] and OpenStack.[5] This implementation allows us to demonstrate the feasibility and soundness of Tornado.

This paper is structured as follows. Section 2 presents our motivation. Section 3 introduces fundamental concepts used in the description of our proposal and discusses related work. Section 4 presents Tornado. Section 5 presents our evaluation. Finally, Section 6 concludes the paper and outlines future work.

2 Motivation

In this section, we describe the motivation for Tornado, by highlighting relevant concerns about consistency and quality of D&C specifications.

Experimentation on production-like environments enables DevOps engineers and operators to develop new features and fix faults by performing ad-hoc modifications. There are D&C specifications, such as Terraform templates, that need be updated accordingly. These updates range from low-level configurations, such

[4] https://www.terraform.io (accessed Oct, 2018)
[5] https://www.openstack.org (accessed Oct, 2018)

as opening ports in a firewall, upgrading or downgrading software packages, to structural changes, such as duplicating services or modifying the scaling policies of virtual resources. Failing to propagate these changes to the specifications appropriately leads to configuration inconsistencies.

The state of the practice for D&C testing is based on static analysis and functional tests [15,16,7]. The former provides quick feedback on minor programming mistakes, such as syntax errors. The latter consists of deploying the infrastructure and execution of unit, integration and system tests to determine if the deployed resources and their configuration are adequate. Deploying and redeploying the system and its infrastructure to sandbox environments is resource and time consuming. Furthermore, modifying a specification can adversely impact another. For example, modifying a `hostname` on a network configuration file without appropriately replicating the update to other specifications (*e.g.*, software deployment) will likely cause a connection timeout. This hinders the experimentation process and requires manual inspection and debugging. Bugs may not appear prior to deployment because specifications are not usually connected, unless they are input to a common compiler/interpreter. Run-time modelling[6] seems to be a feasible alternative to capture the notations' domain logic and validate them prior to deployment, reducing the cases in which the infrastructure must be deployed for testing purposes. It also reduces the developer's cognitive load, as feedback is provided in a timely fashion.

3 Fundamentals and Related Work

This section introduces fundamental concepts for describing our framework and presents related research.

3.1 Round-Trip Engineering

Round-Trip Engineering (RTE)[7] is the process of ensuring the consistency of multiple, changing and interconnected software artefacts [10,11,12,18]. These artefacts participate in a source-target relationship, in which a derivation process creates the target from the source artefact. Target artefacts are usually further altered due to maintenance work or changing requirements [12]. Therefore, these artefacts may no longer be the result of the derivation process and, thus, creating inconsistencies when source artefacts are modified and the derivation process is applied again. RTE ensures consistency between these artefacts by reflecting changes to the target artefact back to the source artefacts.

RTE is closely related to Forward and Reverse Engineering (FE and RE, respectively). FE is the process of deriving one or more target artefacts from one or more source artefacts. RE is the process of reconstructing these sources from the target artefacts, recovering any information lost in the derivation process [19,10].

[6] In the literature often referred to as models@run.time [17]

[7] Model synchronisation and RTE are often used interchangeably in the literature [12]

3.2 Continuous Integration

Continuous Integration (CI) is an agile software engineering practice that allows developers to frequently merge work to a shared mainline multiple times per day [20,13]. It includes frequent automated building and testing of the software in response to code modifications. A typical implementation of this practice includes a CI server that pulls code from a version control repository and executes interconnected steps to compile the code, run unit tests, check quality and build deployable artefacts. Even though automating the integration process is important for adoption, the relevance of CI lies in the frequency of integration. It has to be regular enough to provide quick feedback to developers, thereby improving their productivity and the software quality [20]. CI has the effect of producing shorter release cycles.

3.3 Infrastructure As Code

Infrastructure as Code (IaC) is an approach to provisioning and managing dynamic infrastructure resources through machine-readable configuration files [15,16,7]. It is also referred to as programmable infrastructure in reference to the adaptation and application of practices and tools from software engineering on IT infrastructure management. As a result, changes to the computing infrastructure and/or the execution environment are made in a structured way, by means of reliable and established processes [7]. The benefits of IaC include repeatability of creating and configuring execution environments, management automation, development agility and infrastructure scalability [16].

3.4 Related Work

Software deployment is specified using semi-formal graphical notations, informal diagrams, scripts, domain-specific languages (DSLs), and modelling languages.

The *UML deployment diagram* is a well-known notation that provides a graphical language to describe a static representation of a system's architecture. Though it has been refined in several versions of the UML standard, it continues to be one of the least adopted diagrams among UML users [21] and within the model-driven engineering (MDE) community [22]. This diagram allows to specify only a portion of the required system elements (*e.g.*, infrastructure provisioning, network configuration and elasticity requirements [23,24]). UML lacks the semantics for translating deployment diagrams into code, therefore existing transformation approaches limit the diagram semantics and the supported technologies.

Wettinger *et al.* [25] propose i) a methodology to implement the DevOps paradigm in practice with a high degree of automation; and ii) DevOpslang, a DSL to deploy cloud applications. The purpose of DevOpslang is to bridge the gap between developers and operators by supporting the proposed methodology. Nevertheless, the automation considered in its design seems incomplete with respect to its motivation: it only considers forward engineering, from development to operations, leaving out the continuous cycle as advocated in DevOps. Thus, offering no support at run-time.

Thiery *et al.* [26] address the problem of providing testers with an automated and provider-independent method to deploy and test cloud applications. They define a DSL that allows testers to describe how an application is deployed, and which cloud resources are required and available for the deployment. The DSL generates a set of provider-specific commands based on the providers' command-line applications. The authors claim to support a re-deployment scenario in their evaluation, however, it is rather a deployment to a new cloud platform (*i.e.*, a new deployment). The proposed DSL does not consider any kind of support once the application has been deployed.

Glaser [27] proposes a model-driven and topology-based framework that generates concrete deployment instances compliant with TOSCA. These instances are derived based on a domain model specification whose parameters change over time. The proposed framework updates the running infrastructure on user demand. To achieve this, Glaser proposes a DSL to map domain modelling parameters to parameters of the cloud infrastructure. The proposed DSL implements forward engineering only, providing no support on the operations side.

Holmes [28] proposes MING, a model and view based framework for describing and deploying cloud data centres. MING separates concerns into different views, namely, inventory, networking and configuration, allowing stakeholders to relate to concerns that are relevant to them. These views are realized by an OpenStack-tailored DSL. MING allows to adapt an already deployed data center, either adding new resources or providing software upgrades. As for the aforementioned works, MING realizes forward engineering only.

Significant work has been done in automating the processes to integrate development and operations better. IaC plays an important role in this effort, as it enables the application of software engineering practices to infrastructure design and management. However, there are still many opportunities to strengthen the linkage between both sides of the DevOps development cycle. Moreover, emerging practices, such as continuous experimentation and feedback, require standard and automated processes to integrate run-time data back into development.

4 Tornado: A Framework for RTE in DevOps

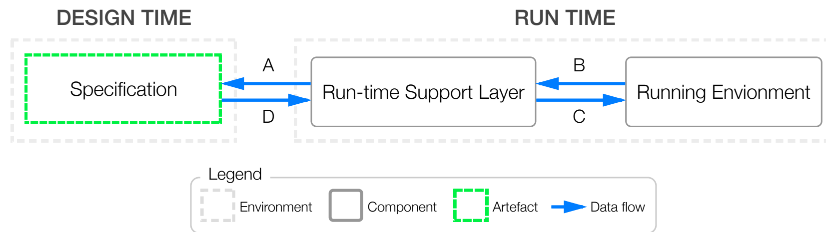

Fig. 1. High level overview of Tornado

The design-time artefacts supported by Tornado are text-based, structured specifications. Our framework reconciles these specifications with their corresponding elements from the running environment. To do so, we introduce a run-

time support layer that bridges D&C work from development and operations. This layer contains models at run-time (MARTs) that represent the elements from the running environment. These models are eventually transformed into text to keep the specifications updated. Figure 1 depicts a high-level overview of our framework. It shows how information flows between design- and run-time through the run-time support layer.

We adhere to the MART definition proposed by Bencomo *et al.* [29]:

> *An MART can be defined as an abstract representation of a system, including its structure, behaviour and goals, which exists in tandem with a given system during the actual execution time of that system [...]*

Tornado is based on Castañeda's operational framework [30]. This framework comprises four main components: a notation-model mapping, a catalogue of operations to update the model's instances, the run-time semantics from the application domain, and causal links. The latter are used to propagate changes among the models that are connected, as in the example presented in Section 2 about a software deployment specification associated with a network configuration file. The models associated with these specifications must be causally connected as follows: the software model references a host name defined in the network model; when the latter changes, the change is propagated to the former, so it is updated accordingly.

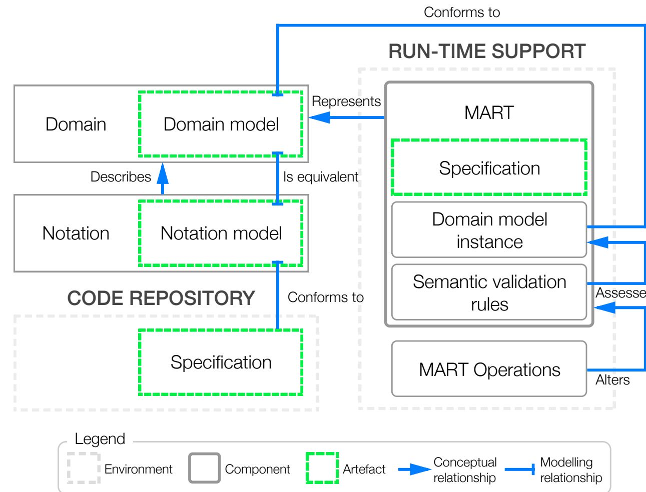

Fig. 2. Tornado's concepts

Figure 2 depicts what an MART is in terms of its internal components. In Tornado, an MART is not only a model instance but a 3-tuple containing a model instance, a specification instance (*i.e.*, one or more files) and a set of semantic validation rules. The model and specification instances are kept in sync automatically. The validation rules check the quality of the model to guarantee its integrity. For example, a computing resource may be given an IP address

outside its subnet range. A simple validation rule can discover this mistake, avoiding the deployment of the whole infrastructure, offering quicker feedback and spending fewer resources. Furthermore, these validations can be delegated to other software components. In our example, the network configuration verification can be delegated to simulation engines or network virtualisation platforms, which can be run in memory without the need for deploying more resources.

As shown in Figure 2, an MART is associated with a set of operations. Each operation contains the run-time semantics to alter the model instance. An operation is associated with a set of Pre- and Post-validation rules, which check the state of the model before and after altering it.

The relationship between a specification and an MART is detailed in Figure 2. A MART conforms to a domain model, and must be equivalent to a notation. That is, Tornado expects a one-to-one relationship between the specification notation and its corresponding model. However, achieving such a relationship is often difficult; it may be necessary to limit the facts that can be expressed with the notation to guarantee said equivalence. To map the concepts from one model to another, the pair model-specification is associated with a set of transformations. For example, an MART representing the networking domain can be set up to work with OpenStack HOT[8] and/or HCL[9] (*i.e.*, Terraform templates' notation). Each of these configurations knows how to update the model instance and the specification file, given a change in either of them.

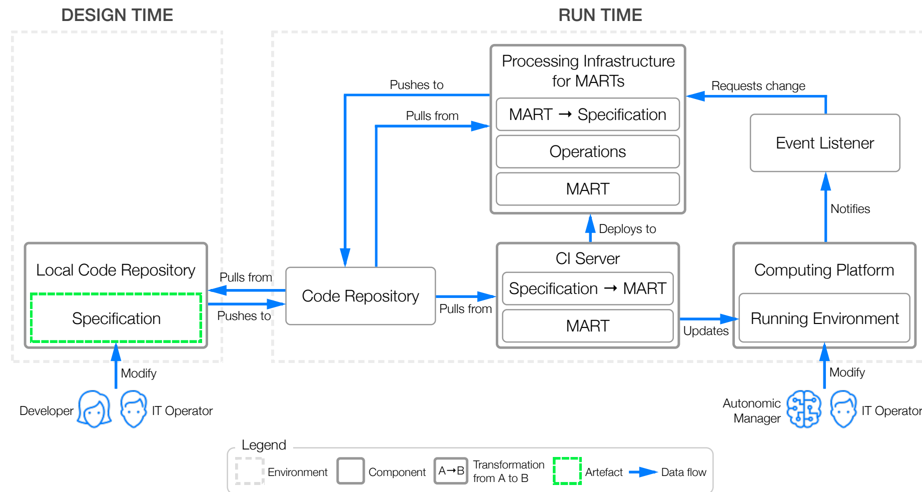

Fig. 3. Continuous Integration loop in Tornado

Next, we describe the arrows `A`, `B`, `C` and `D` from Figure 1. These arrows are later refined in Figure 3.

[8] https://docs.openstack.org/heat/latest/template_guide (accessed Oct, 2018)
[9] https://www.terraform.io/docs/configuration/syntax.html (accessed Oct, 2018)

A: Specification → MART

This interaction is initiated by a developer. Once she pushes changes to the version control repository, a CI server temporarily instantiates the corresponding MART based on the current version of the specification. If it passes the quality checks and the MART is already deployed to the run-time support layer, the existing MART is updated and the CI server proceeds to apply the changes to the running environment. In case the MART is being instantiated for the first time, a new instance is deployed.

B: MART → Running Environment

This interaction is not further explored in this paper, given our motivation to integrate operations work in the opposite direction. Nevertheless, an MART may update a running environment as part of the change propagation chain from a causal connection, as described above.

C: MART ← Running Environment

This interaction is initiated by a change in the running environment. A listener catches an event propagated by the supporting platform and initiates a procedure to update the MART instance accordingly.

D: Specification ← MART

This interaction is initiated when an MART instance is updated by the running environment it represents. A procedure is triggered to transform the instance to the corresponding specification notation. The resulting text is used to update the remote file in the version control repository.

Figure 3 depicts Tornado's continuous integration loop. This loop extends the concept of CI to frequently integrate changes into a running environment into development. D&C specifications are usually treated in the same way as application code. This traditional use of CI only considers integrating work at the request of developers. It means that any kind of manual work would require an operator to remember and translate data from one tool to another, from one syntax to another, and possibly from one paradigm to another (*e.g.*, imperative to declarative). The continuous integration loop we propose automates that process. Furthermore, DevOps engineers and operators are not the only actors who modify a running environment. Autonomic managers have already assumed a significant role in understanding run-time operations. Dynamic scaling policies, for example, automatically scale computing resources in response to changing service demand. The actions of these autonomic managers are not generally reflected in the specifications.

In our proposal, the CI server deploys the model directly to the processing infrastructure instead of updating an already deployed model one event at a time.

4.1 CI Considerations

This subsection discusses three main CI considerations regarding the implementation and adoption of Tornado. We outline concerns that could potentially affect the development workflow, and propose alternative solutions.

C1: Contribution Model. Tornado enables the run-time support layer to make code contributions. Although CI provides mechanisms to guarantee quality, unsupervised changes can produce adverse effects. This can happen, for example, due to an operation mistake or a bug in the run-time semantics associated with a model. In addition to the *committer* model, in which the run-time support layer is added as a collaborator to the repository (*i.e.*, it is granted write access), we propose the *contributor* model, in which code modifications are proposed as pull requests rather than committed directly.

In the case of the committer model, there would be no delay in reflecting the changes in the specifications. For this reason, this model would likely produce fewer merge conflicts. However, it does not mitigate the risk of unwanted side-effects. In the case of the contributor model, the risk is completely avoided. Nevertheless, additional time must be allocated to review the pull requests, delaying the update and increasing the possibility of merge conflicts. While a pull request remains open, the MART instance is inconsistent with respect to the specification or the running environment.

It is common today that computing platforms and autonomic managers make decisions to affect a running system. Therefore, it is acceptable, at least in some cases, to grant commit access to the run-time support layer. We believe that providing both contribution alternatives is the best option.

C2: Conflict Resolution. Conflict resolution is not a trivial task. It requires spending time inspecting the code and making informed decisions about the merging conflict(s). Therefore, automating conflict resolution requires simplifying the problem. We suggest two strategies to do so. First, we propose to avoid conflicts related to formatting. The transformation from MART to specification must follow a standard process, which always generates statements in the same order, case and format (*e.g.*, spacing and indentation). To facilitate following these measures in development, we propose to use a formatting utility before committing changes. And second, we propose to give priority to one of the actors (*e.g.*, the run-time support layer). In case of merge conflicts, the run-time support layer can decide to either drop the local changes or replace the remote ones, according to its assigned priority level. The former requires to rollback the latest changes to keep the MART instance consistent with the remote specification.

C3: Quality Assurance. One of the most important parts of CI is the continuous application of quality control. Tornado re-uses the concept of pre- and post-validation rules from Castañeda's operational framework [30] to ensure quality conditions before and after modifying the model. However, there may

be concerns regarding the model itself, rather than its state with respect to a certain operation. For instance, referring to the example above about the IP address outside of range. The state itself is erroneous, making it necessary to check its quality before deployment. There are also other kinds of concerns related to business restrictions; validations on the model instance allows, for example, to limit what can be deployed by the tenant. These business restrictions can be implemented as semantic restrictions on the model, as the model itself represents entities from the rules' domain.

5 Evaluation

In this section, we present a proof of concept implementation of TORNADO. This implementation covers all the topics discussed in Section 4, including the CI considerations. The source code of this implementation is available in a Github repository.[10]

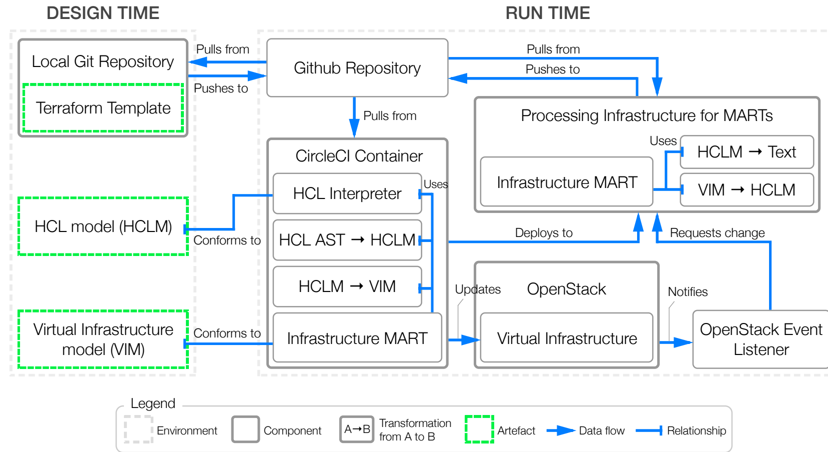

Fig. 4. Evaluation Setup

Figure 4 depicts the evaluation setup. We chose the IaC tool Terraform and the OpenStack platform to realise this proof of concept. Consequently, the notation specification is HCL (*i.e.*, Terraform templates' notation) and the running environment is a virtual infrastructure. Figures 5 and 6 represent the HCL and virtual infrastructure models, respectively. Notice that these models are limited with respect to the entities they represent. However, they are complex enough to demonstrate the usefulness and soundness of this framework.

The HCL and infrastructure models were developed using the Eclipse Xcore project[11], and the model transformations using the Xtend language[12]. The HCL interpreter was developed using Eclipse Xtext[13].

---

[10] `https://github.com/RigiResearch/jachinte-DevOps2018-evaluation`
[11] `https://wiki.eclipse.org/Xcore` (accessed Oct 2018)
[12] `http://www.eclipse.org/xtend` (accessed Oct 2018)
[13] `http://www.eclipse.org/Xtext` (accessed Oct 2018)

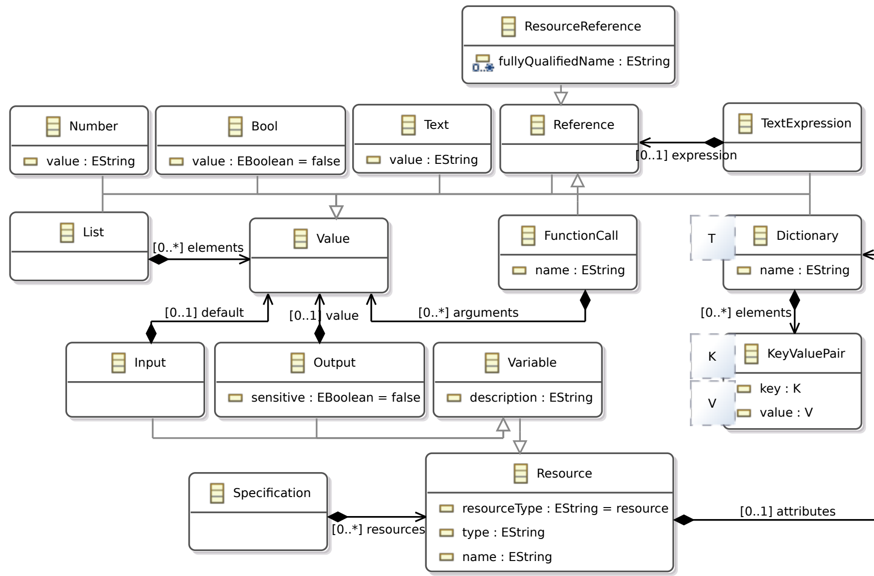

Fig. 5. HCL Model

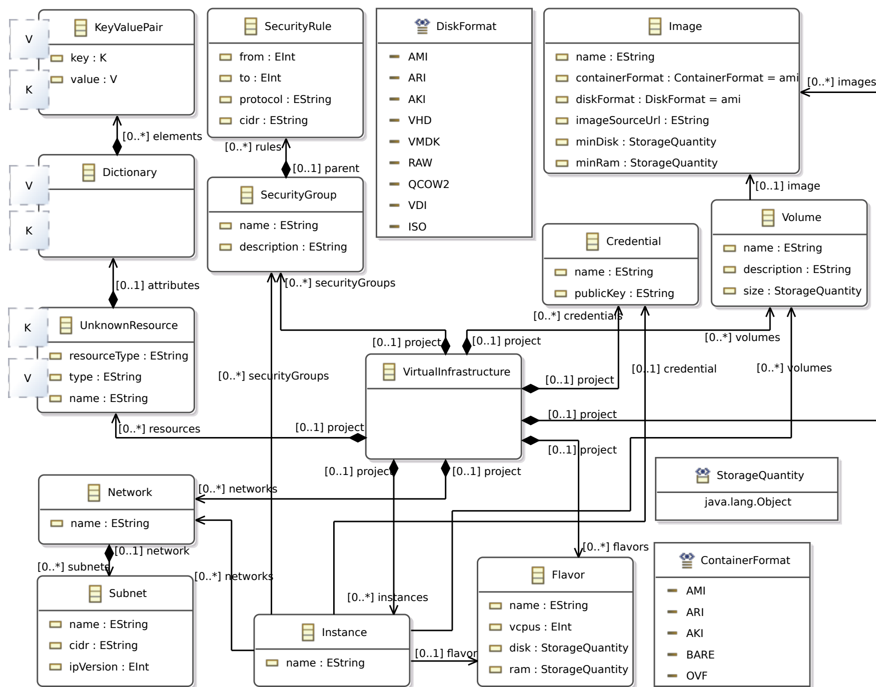

Fig. 6. Virtual Infrastructure Model

Next we describe the main components of the evaluation setup and describe the development workflow associated with this implementation.

Infrastructure MART This MART is composed of a Terraform specification, an instance of the Infrastructure model and a set of supported operations and validations on the model instance. The specific operations supported at the time of writing this paper are: adding a new resource and removing an existing one. Supporting more operations requires implementing the interface `Operation`. As an example, we added one semantic validation that constraints the RAM size of any virtual machine instance to be launched. Adding more validations is possible by implementing the interface `Rule`. The MART ensures that the model instance and the specification are always in sync, so any modification to one another causes a synchronization process and a commit if necessary. When the specification changes, the Terraform state is updated consequently. Before committing, the MART invokes the Terraform format command. Commit messages end with "`[skip ci]`" to avoid unnecessary CI builds. However, CI could be used to reinforce quality policies on operations (manual) work.

Processing Infrastructure for MARTs This component offers a service to register or update an MART and execute operations on it. Registering a new MART causes the corresponding code repository to be cloned. Updating an existing MART causes its corresponding repository to be updated accordingly.

CircleCI Container This component uses the HCL interpreter to create an Abstract Syntax Tree (AST) out of a Terraform template. It uses the AST to instantiate the HCL model and then transforms it to an instance of the virtual infrastructure model. Then, it instantiates the Infrastructure MART and executes the semantic validations associated with it. If the MART passes the validations, it is deployed to the processing infrastructure for MARTs and the resources specified in the Terraform template are deployed. If the validations fail, the developer is notified. This workflow is specified in a YAML[14] configuration file containing three jobs: `validate_terraform`, `deploy_models` and `deploy_terraform`.

5.1 Development Workflow

The development workflow is as follows. An operator creates a Terraform template using OpenStack as cloud provider. When the template is pushed to the Github repository, the CI server pulls the template and creates a local instance of the Infrastructure MART based on the template. If the instance passes the semantic validations (*i.e.*, there are no instances violating the RAM constraint), the MART is deployed to the processing infrastructure for MARTs. Then, the CI server deploys the Terraform template, updating the OpenStack resources. The event listener consumes the events generated by OpenStack but ignores them, as these changes are authored by a known user name associated with the CI server

[14] `http://yaml.org` (accessed Oct 2018)

and the changes have been already applied to the model. At this point, the template and the OpenStack resources are in sync. When the operator modifies the OpenStack resources (*i.e.*, creates or removes resources using the Horizon dashboard or the CLI client), the event listener gets a notification. It then executes the corresponding operation on the MART by means of the processing infrastructure for MARTs. The MART adapts accordingly, committing and pushing any modification to the template to the Github repository.

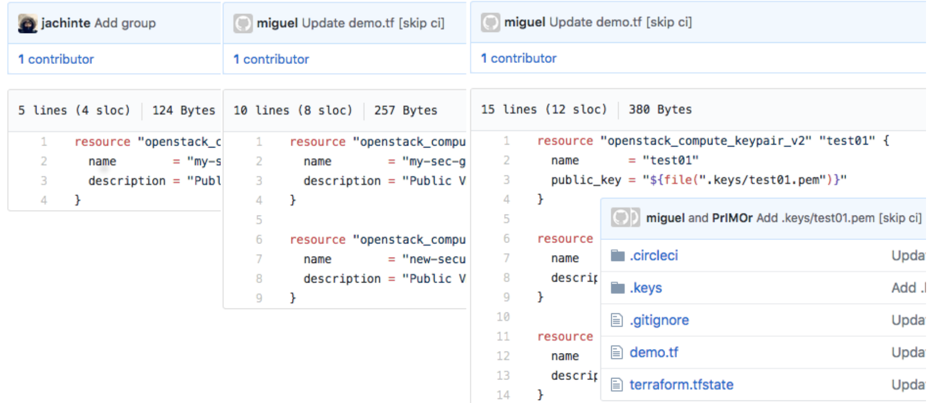

Fig. 7. Updates to the Terraform template on Github

Figure 7 depicts three screenshots taken from the test Github repository. The first one displays the initial state of the template, as created by the developer (*i.e.*, user *jachinte*). The second and third screenshots display the template after the processing infrastructure (*i.e.*, PrIMoR on behalf of user *miguel*) committed changes. In the last case, a new file is added and referenced from the template.

6 Conclusions and Future Work

There is a lack of standard processes and tools in DevOps to integrate operation information back into development readily. In this paper, we focused on the lack of automation support for updating D&C specifications from manual changes to experimental setups. DevOps engineers and operators are in charge of keeping these specifications consistent, remembering every change and translating them from one syntax and paradigm to another. We presented TORNADO, a two-way CI framework (*i.e.*, Dev $\xrightarrow{\text{CI}}$ Ops and Dev $\xleftarrow{\text{CI}}$ Ops) that keeps D&C specifications always in sync with the systems they configure and deploy. We evaluated TORNADO by implementing a proof of concept based on Terraform templates and the OpenStack platform, demonstrating its feasibility and soundness.

Although TORNADO focuses on RTE for D&C, it potentially enables further synchronisation between other design artefacts (*e.g.*, architecture design) and the D&C specifications, completing the continuous development loop. This would provide operators and autonomic managers with a standard mechanism

to contribute to the evolution of the system. That is, their actions would directly affect the development artefacts too.

Acknowledgments. This work was funded in part by the National Sciences and Engineering Research Council (NSERC) of Canada, IBM Canada Ltd. and IBM Advanced Studies (CAS), the University of Victoria (Canada), and Universidad Icesi (Colombia).

References

1. Sharma, S., Coyne, B.: DevOps for Dummies. Limited IBM Edition (2013)
2. Zhu, L., Bass, L., Champlin-Scharff, G.: DevOps and Its Practices. IEEE Software **33**(3) (2016) 32–34
3. Fagerholm, F., Guinea, A.S., Mäenpää, H., Münch, J.: Building Blocks for Continuous Experimentation. In: Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering. RCoSE 2014, New York, NY, USA, ACM (2014) 26–35
4. Shahin, M., Babar, M.A., Zhu, L.: The intersection of continuous deployment and architecting process: Practitioners' perspectives. In: Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. ESEM 2016, New York, NY, USA, ACM (2016) 44:1–44:10
5. Fabijan, A., Dmitriev, P., Olsson, H.H., Bosch, J.: The evolution of continuous experimentation in software product development: From data to a data-driven organization at scale. In: 2017 IEEE/ACM 39th International Conference on Software Engineering. ICSE 2017 (2017) 770–780
6. Schermann, G., Cito, J., Leitner, P.: Continuous experimentation: Challenges, implementation techniques, and current research. IEEE Software **35**(2) (2018) 26–31
7. Morris, K.: Infrastructure As Code: Managing Servers in the Cloud. 1st edn. O'Reilly Media, Inc. (2016)
8. Spanoudakis, G., Zisman, A. In: Inconsistency Management in Software Engineering: Survey and Open Research Issues. World Scientific Publishing Company (2012) 329–380
9. Kruchten, P., Nord, R.L., Ozkaya, I.: Technical debt: From metaphor to theory and practice. IEEE Software **29**(6) (2012) 18–21
10. Henriksson, A., Larsson, H.: A Definition of Round-Trip Engineering. Technical report (2003)
11. Sendall, S., Küster, J.: Taming model round-trip engineering. In: Proceedings of Workshop on Best Practices for Model-Driven Software Development. (2004) 1
12. Hettel, T., Lawley, M., Raymond, K.: Model Synchronisation: Definitions for Round-Trip Engineering. In Vallecillo, A., Gray, J., Pierantonio, A., eds.: Theory and Practice of Model Transformations, Berlin, Heidelberg, Springer Berlin Heidelberg (2008) 31–45
13. Shahin, M., Ali B., M., Zhu, L.: Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. IEEE Access **5** (2017) 3909–3943
14. : ISO/IEC/IEEE International Standard - Systems and software engineering – Vocabulary. ISO/IEC/IEEE 24765:2010(E) (2010) 1–418
15. Hüttermann, M. In: Infrastructure as Code. Apress (2012) 135–156

16. Nelson-Smith, S.: Test-Driven Infrastructure with Chef: Bring Behavior-Driven Development to Infrastructure As Code. O'Reilly Media, Inc. (2013)
17. Blair, G., Bencomo, N., France, R.B.: Models@run.time. Computer **42**(10) (Oct 2009) 22–27
18. Rahm, J., Graube, M., Urbas, L.: A proposal for an interactive roundtrip engineering system. In: 2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA). (Sept 2017) 1–7
19. Tilley, S.R., Wong, K., Storey, M.A.D., Müller, H.A.: Programmable Reverse Engineering. International Journal of Software Engineering and Knowledge Engineering **04**(04) (1994) 501–520
20. Fitzgerald, B., Stol, K.J.: Continuous software engineering: A roadmap and agenda. Journal of Systems and Software **123** (2017) 176–189
21. Petre, M.: UML in Practice. In: Proceedings 35th International Conference on Software Engineering. ICSE 2013, Piscataway, NJ, USA, IEEE Press (2013) 722–731
22. Hutchinson, J., Rouncefield, M., Whittle, J.: Model-driven Engineering Practices in Industry. In: Proceedings 33rd International Conference on Software Engineering. ICSE 2011, New York, NY, USA, ACM (2011) 633–642
23. Inzinger, C., Nastic, S., Sehic, S., Vögler, M., Li, F., Dustdar, S.: MADCAT: A Methodology for Architecture and Deployment of Cloud Application Topologies. In: Proceedings 8th International Symposium on Service Oriented System Engineering. SOSE 2014, Oxford, UK (2014) 13–22
24. Copil, G., Moldovan, D., Truong, H.L., Dustdar, S.: SYBL: An Extensible Language for Controlling Elasticity in Cloud Applications. In: Proceedings 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. CCGrid 2013 (2013) 112–119
25. Wettinger, J., Breitenbücher, U., Leymann, F.: DevOpSlang - Bridging the Gap between Development and Operations. In Villari, M., Zimmermann, W., Lau, K.K., eds.: Service-Oriented and Cloud Computing. Volume 8745. Springer Berlin Heidelberg (2014) 108–122
26. Thiery, A., Cerqueus, T., Thorpe, C., Sunyé, G., Murphy, J.: A DSL for Deployment and Testing in the Cloud. In: Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation Workshops. ICSTW 2014, IEEE Computer Society (2014) 376–382
27. Glaser, F.: Domain Model Optimized Deployment and Execution of Cloud Applications with TOSCA. In Grabowski, J., Herbold, S., eds.: System Analysis and Modeling. Technology-Specific Aspects of Models. Volume 9959. Springer International Publishing (2016) 68–83
28. Holmes, T.: Ming: Model- and View-Based Deployment and Adaptation of Cloud Datacenters. In Helfert, M., Ferguson, D., Méndez Muñoz, V., Cardoso, J., eds.: Cloud Computing and Services Science. Volume 740. Springer International Publishing (2017) 317–338
29. Bencomo, N., Bennaceur, A., Grace, P., Blair, G., Issarny, V.: The role of models@run.time in supporting on-the-fly interoperability. Computing **95**(3) (Mar 2013) 167–190
30. Castaneda, L.: Runtime Modelling for User-Centric Smart Applications in Cyber-Physical-Human Systems. PhD Thesis, Department of Computer Science, University of Victoria (2017)