POJO Model Representation and Policy-Action Reconfiguration of Kubernetes Cloud Systems

Final Progress Report

Samar G. Sajnani

Computer Science Thesis (CS4490Z)

Department of Computer Sceince

Western University

3/29/19

Supervisor: Dr. Kostas Kontogiannis

Course Instructor: Dr. Nazim Madhavji

Terminology:

Software dynamic reconfiguration

A software system is made of different functional components, these components can be added, removed, and manipulated based on software needs. These operations comprise reconfiguration steps. If reconfiguration steps can occur without software downtime, the reconfiguration is dynamic in nature.

Metamodel

Metadata for model generation, defines the classes and types of components that can be used to make a system of objects. Objects are generated instances based on the classes and types of the metamodel. Examples include Java class-based metamodels that are used by Java to create model instances defined as Plain Old Java Objects (POJOs).

Policy-action

A trigger system based on Boolean statements that are defined as a policy, when the statements are all satisfied the respective action is triggered.

Plug-And-Play Architecture

Functionality of a program is divided into separated functional modules that can be developed independently. Modules can be interchanged with new modules without having to overhaul the entire architecture.

Acronyms

API - Application Programmer Interface

EMF - Eclipse Modelling Framework

POJO - Plain Old Java Object

MDD - Model-Driven Development

Abstract

[Context and motivation]

In 1998, a paper demonstrated that a distributed system could use a computational method to generate dynamic reconfiguration brought on by dynamic system change [1]. Dynamic reconfiguration can be further abstracted using Model-Driven Development (MDD). Recent studies of MDD have shown that it can be used to reduce code redundancy and development costs, however, MDD-MSA systems are underutilized [4].

[Question/Problem]

Combining MDD with a policy-action trigger would be useful to reap the benefits of MDD in MSA. However, the lack of granular models has plagued the development of such systems.

[Principal ideas/results]

A dynamic reconfiguration system that uses a Plain Old Java Objects (POJOs) model was developed since the native Kubernetes Java Client generates plain old java objects (POJOs) for every resource in a Kubernetes system.

[Contribution]

Three main use cases were described: creating policies and actions for local variables, system metrics, and for external variables. The approach in this study is to satisfy a general-purpose solution to dynamic reconfiguration of Kubernetes cloud systems with a granular model.

1. Introduction

In this section, a summary of dynamic reconfiguration, policy-action systems, model-driven development, Docker, Kubernetes, and project-related information will be provided. Software dynamic reconfiguration is the process of updating a program while maintaining its active state [1]. Dynamic reconfiguration has the potential to reduce costs of software development because of the automation of reconfiguration steps [2]. Automated dynamic reconfiguration systems are often implemented using policies and actions. A policy represents a Boolean statement that results in an accepting (true) state or a violating (false) state. If the policy transitions from one state to another it may enact some action as a response to the state change [3].

While creating policies and actions is useful, it can result in teams producing redundant types of action and policy conditions. One way to reduce software redundancies is to use a Model-Driven Development (MDD) approach. In MDD, a metamodel holds the blueprints for a component in a system. To model a system, the metamodel is used to generate instances of objects that match the components on the system, making an object-based copy of the system, known as a model. These concepts are the basis of MDD, and since a metamodel is used to generate the system, the same metamodel can contain reusable code to reduce redundancies [4].

In practice, cloud systems offer a practical approach to software scalability and high-availability. Many cloud software environments are based on Microservice architectures (MSAs). Most commonly, MSAs use Docker as a software containerization tool that generates virtual environments by using OS-level virtualization. Images in Docker hold the metadata of Docker containers, once the images are initiated the Docker image runs a Docker container. Another tool called Kubernetes also interfaces with Docker to orchestrate the initiation of images as microservices across multiple connected computing systems.

Since Kubernetes cloud systems provide highly-available services they cannot have down-time during reconfigurations. As a result, reconfiguration of cloud systems is carried out dynamically. Often these dynamic reconfigurations result in redundancies across teams. To combat these redundancies, a recent approach of dynamic reconfiguration of a cloud software system using MDD has been gaining traction [4]. However, the metamodels in literature that describe the MDD of Kubernetes cloud systems lack granularity, because they are not native to Kubernetes [5]-[6]. Also, the process of MDD for MSA is an underdeveloped area of research [4]. Therefore, the purpose of this project was to develop a POJO-based MDD approach for MSA that will be used as a basis for the dynamic reconfiguration actions of a Kubernetes cloud system. This combination of systems is novel and rather unexplored.

To create a POJO-model the Kubernetes Java Client was used as it was native to the Kubernetes ecosystem. A variable fetcher was implemented to gather microservice, system, or externally related variables. Finally, a policy-action system was created composed of policy and action planes. Three main integration tests were conducted to validate three specific use cases related to interdependencies, metrics, and upgrades. The significance of this research is that it is general, focused on reducing redundancies, and it has the potential to describe a Kubernetes cloud system

in its' entirety. The following sections will provide a background of the literature, objectives, methodologies, results, discussions and conclusions.

2. Background

The concept of dynamic reconfiguration for cloud systems is not a novel idea. The implementations of dynamic reconfiguration have been studied in software architectures, graph transformation, software adaptation, metamodeling, and reconfiguration patterns [7].

2.1. Analyzing Dynamic Change in Distributed Software Architectures

- One of the first examples of dynamic configuration was in a 1998 paper by Jeff Kramer and Jeff Magee
- It described the use of a four-step model for creating a dynamic change in a distributed software system
    1. Identifying elements of an arbitrary active state of the software system
    2. Converting the arbitrary active state to a passive state
    3. Performing the changes needed to the systems' passive state
    4. Activating the elements from their passive state
- Operations such as disconnecting, deleting, creating, and connecting were used as examples [1]

2.2. Dynamic Reconfiguration for Cloud Systems

- In 2016, Prieto *et al.* studied the generation of cloud services using an Ecore metamodel
- This model was used by developers to specify a new or updated service and generate the reconfiguration scripts that carry out the service change [5]

- In January of 2017, Düllman and Hoorn demonstrated a model-driven dynamic reconfiguration of microservice architectures (MSAs) using a simple Ecore metamodel with Kubernetes
- Dynamic reconfiguration events were triggered based on performance anomaly detection
- The authors suggested that future work should be targeted to analyze more complex microservice environments and to model extraction from a running Kubernetes environment [6]

2.3. Importance of Policy-Action Systems

- Amazon Web Services uses policies to maintain high performance and scalability of their cloud services [3]
- A patent from 2015 shows that the use of actions and policies is important for dynamic reconfiguration of multi-user systems especially in autoscaling cloud networks depending on the user base size [8]

2.4. Model-Driven Development (MDD)

- In May of 2018, Sorgala *et al.* outlined that microservice architectures consist of multiple, loosely coupled microservices
- Normally a single Development Operations (DevOps) team is responsible for each microservice, however, crafting of microservices across dispersed teams can lead to redundant implementations
- An emerging way to deal with such redundant complexity is to use a model-driven approach across multiple teams
- Added that MDD-MSA is underdeveloped while MDD on cloud infrastructure is quite widespread [4]

2.5. Analysis and Research Gap

Kramer and Magee demonstrated one of the first theoretical methods of distributed software reconfiguration in 1998 [1]. Their method spawned a plethora of further research on this topic, however, the focus of software-related dynamic reconfiguration has mostly been theoretical. Out of the minimal application-based studies, only a few are related to cloud architecture and even fewer are related to metamodel-based reconfiguration. The usage of MDD for MSAs is an emerging technique to combat the redundancy of loosely coupled microservices of Kubernetes cloud systems [4]. Research by Prieto *et al.*, in 2016, was one of the first examples of a model-based dynamic configuration of a cloud system. However, although their dynamic reconfiguration steps were automated using scripts, the user was responsible for triggering the event [5]. As a result, the system was not completely automated. Action-policy systems can be used to automate the trigger process, as demonstrated by their uses with Amazon Web Services (AWS) [3]. Another notable study was performed by Düllman and Hoorn, in 2017, which demonstrated dynamic reconfiguration based on dynamic triggers for a model-based cloud system. However, their model system was quite rudimentary, as it did not model the Kubernetes system completely, and it only worked well with model generation, not model extraction. There was also no generalized approach as their system was targeted for the specific case of anomaly detection [6].

The identification of a granular metamodel and generic policy-action system for a Kubernetes cloud system is the focus of this project. There is a lack of a highly granular metamodel that can be used to describe Kubernetes cloud systems [5]-[6]. Additionally, most policy-action systems for software cloud systems are focused on performance metrics and do not provide the user with the freedom to create policies and actions for generic cases [6]. Such a system will allow for automation of the trigger and the dynamic reconfiguration process, allowing users to create customized actions based on customized policies.

3. Research Objectives

O1: Identify an appropriate metamodel and generate a model-based representation of a cloud system

O2: Using real-time data from the cloud-system as variables for evaluation in policies

O3: Create policies that will relate the variable data to actions that can be performed on a cloud-system

O4: Scripting an action plan that is responsible for reconfiguring the cloud system in response to a policy

3.1. Significance

Finding a model to describe common Kubernetes-based cloud systems is important for research because the non-physical model can be used to predict or simulate actions. Furthermore, models are used in practice because of their abstraction of cloud elements. If a model could near completely describe a Kubernetes cloud system, it would be useful for many development teams and would reduce costs and redundancies. Policies are significant for this research because they act as a trigger to generate a dynamic reconfiguration event. Creating a dynamic reconfiguration is highly sought after because of its' lack of down-time. Adding model-based actions will be useful in generating a framework that can automate all the different create, read, update and delete (CRUD) actions of a Kubernetes cloud system, allowing the framework to benefit from MDD. The use of a novel policy-action and metamodel Kubernetes cloud reconfiguration system is hypothesized to reduce the development time, cost, and redundancy needed for reconfiguration events by using automation.

4. Methodology

The system architecture for the reconfiguration system described in this paper is composed of different design patterns, technologies, and tools.

4.1. Metamodel

It was important to find a metamodel that could accurately describe a Kubernetes cloud system. After some research the most granular metamodel found was the Native Kubernetes Java Client, this is because the classes of the Java Client describe all the Kubernetes cloud objects, such as Pods and Deployments. The classes can be instantiated to generate Plain Old Java Objects (POJOs) which can then be used to maintain a model of the Kubernetes system. Additionally, the Kubernetes Java Client can be used to perform CRUD operations on Kubernetes cloud resources through its multiple API classes. Finally, the Java Client is released by the Kubernetes development team, therefore, it is expected to describe the Kubernetes ecosystem relatively well and have the newest features incorporated after each new Kubernetes version.

4.2. Design Patterns

For design patterns, the main ones used are the singleton, abstract factory, and adapter patterns. The singleton design pattern is used to statically gather and hold the POJO Model and its object relationships so that the same model can be updated or modified using different instantiations of the POJO Model class. A variation of abstract factory, one that contains all the CRUD operations instead of just a creation method is utilized for the model-based actions. For the real-time variable fetcher, the adapter pattern is used to map the API route (URL) adapter to the variable gathering (Gatherer) adaptee. The adapter pattern is also used to map the policies as adapters to the actions as adaptees [9].

4.3. Tools and Technologies

Docker is used to containerize cloud services and Kubernetes is used to orchestrate these services to the multiple computing units of a cloud system. A Spark server hosts the POJO Model and policy-action system, separately. As for the variable fetcher, the SpringBoot framework hosts a server with the addition of the Java CompletableFutures package to allow for concurrent requests. Last, the Java programming language is used to develop all these different components and to connect them using APIs and REST calls.

5. Results

5.1. Contextual Overview

Many cloud systems today use Kubernetes-based microservice architectures. The idea of a dynamic reconfiguration system involves adding an additional orchestration layer over top of Kubernetes so that sequence of reconfiguration events can be orchestrated based on specific events of the development or production cycle. An example of a dynamic event that requires dynamic reconfiguration is an incremental upgrade as described in Figure 1.

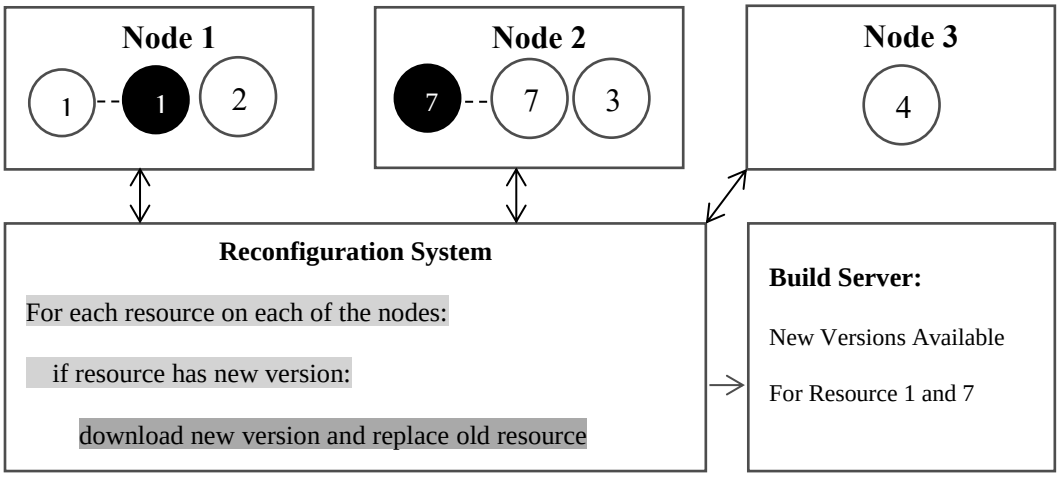

Figure 1 – Example of a Reconfiguration System for Incremental Upgrades. The dashed lines represent transitions from the old to the new resources; the new version is a black filled circle. The policy is highlighted in light gray and the action is highlighted in dark gray.

5.2. Overall Design

The general design of the system developed for this project is summarized in Figure 2.

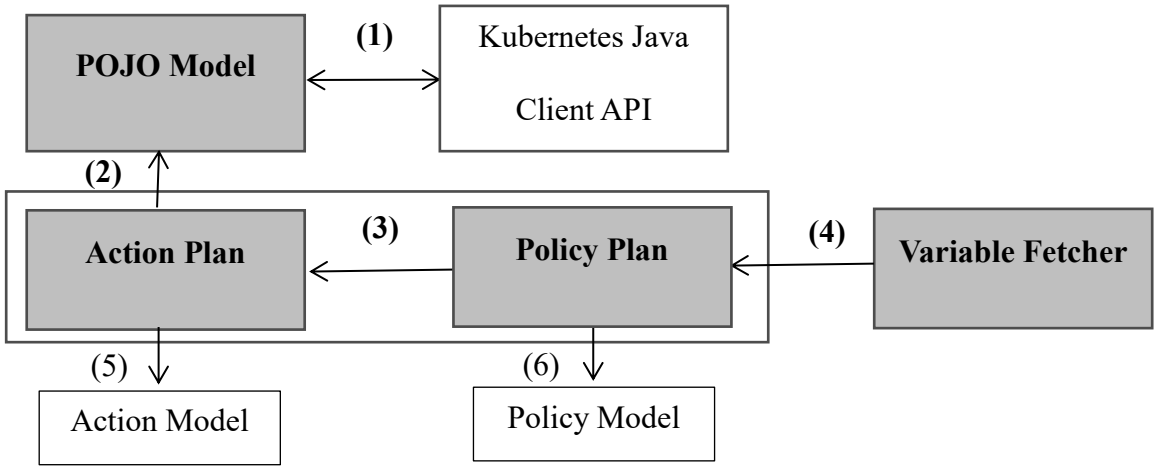

Figure 2 – Components of the Policy-Action Metamodel Reconfiguration System

In Figure 2, There are a total of seven components with six connections between them. Out of the components, four were developed for this project and are highlighted in the gray boxes. These components are the POJO Model, Action Plan, Policy Plan and Variable Fetcher. Additionally, four of the connections were also established, particularly the connections labelled 1, 2, 3, and 4. The Action Model and Policy Model components along with their connections are part of the future work for this project. One of the benefits of using this system is that the components can be developed individually by different teams allowing for abstraction of the dynamic reconfiguration pipeline. There can be domain experts in policy plans, action plans, variable fetching, or POJO Model generation. Additionally, this system provides a framework that is centered on MDD allowing users to create functions on different teams that can be shared between teams to reduce the overall redundancy of loosely-coupled microservices. The use of MDD can also facilitate simulation and time evolution data of the Kubernetes system, as is demonstrated by the routes for difference, triggerReset, and dryRun described in the POJOModel Server. Finally, this system is not limited to performance metrics, the functions for the policies, actions, and variables can all be user defined allowing users to generate multiple different types of dynamic reconfiguration events.

5.3. Plain Old Java Object Model Spark Server

The Kubernetes team supports APIs for multiple different languages, since Java is an Object-Oriented Programming (OOP) language, their implementation of the Kubernetes APIs generates POJOs on most API calls. These POJOs are rarely used to do any actual work. For this project, a

novel technique of collecting the POJOs from API calls is used in the singleton class POJOModelService to generate a Kubernetes cloud model. A complete set of resources with all namespaced and non-namespaced resources was gathered from Kubernetes using the CoreV1API, AppsV1API, AutoscalingV1API, and BatchV1API. A HashMap was used to store resource types as Strings mapped to an ArrayList of the corresponding POJOs. The stored POJOs were used to hold the current state of the resources in the cluster. These resource lists contain limited resource relationship information; therefore, a second data structure was created. This data structure was a HashMap that maps each resource to all its owned resources. Together these two data structures compose the POJO Model.

The POJOModelController is a routing class that allows users to call a REST API to perform CRUD operations on a cloud system and reflect these changes in the singleton POJO Model. The CRUD operations were developed for the Pods, Deployments, Services, and Namespaces and followed a type of Abstract Factory design pattern. Figure 3 shows how these actions were modelled for the Pod resource, the same technique was used to model the actions for Deployments, Nodes, and Services. One of the main advantages of this system over systems described in literature is that it uses the Kubernetes Java Client API. This is beneficial because the Client API contains virtually all the possible functions that can be applied to Kubernetes resources. These functions are especially useful while developing the CRUD operations.

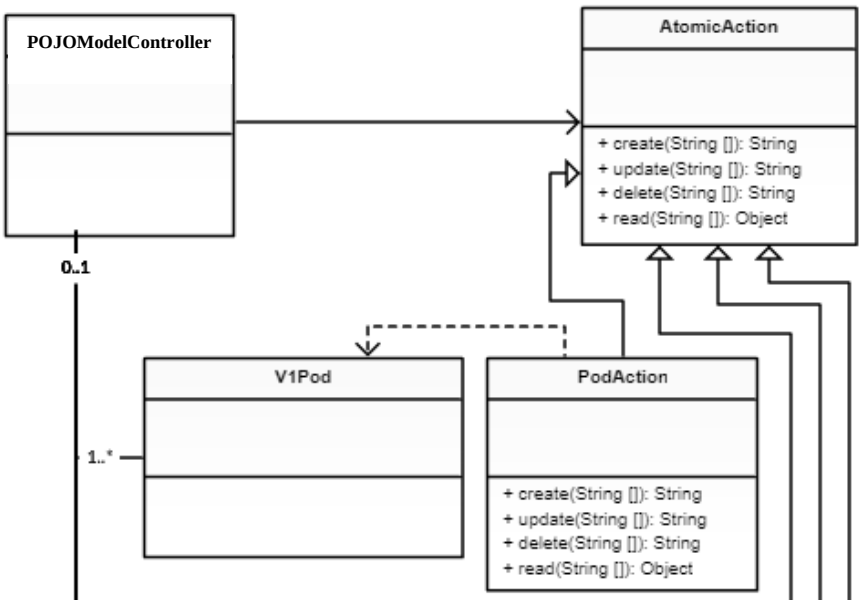

Figure 3 – POJO Model Actions

In Figure 4, the test cases for the CRUD actions on Pods, Deployments, Namespaces, and Services are presented.

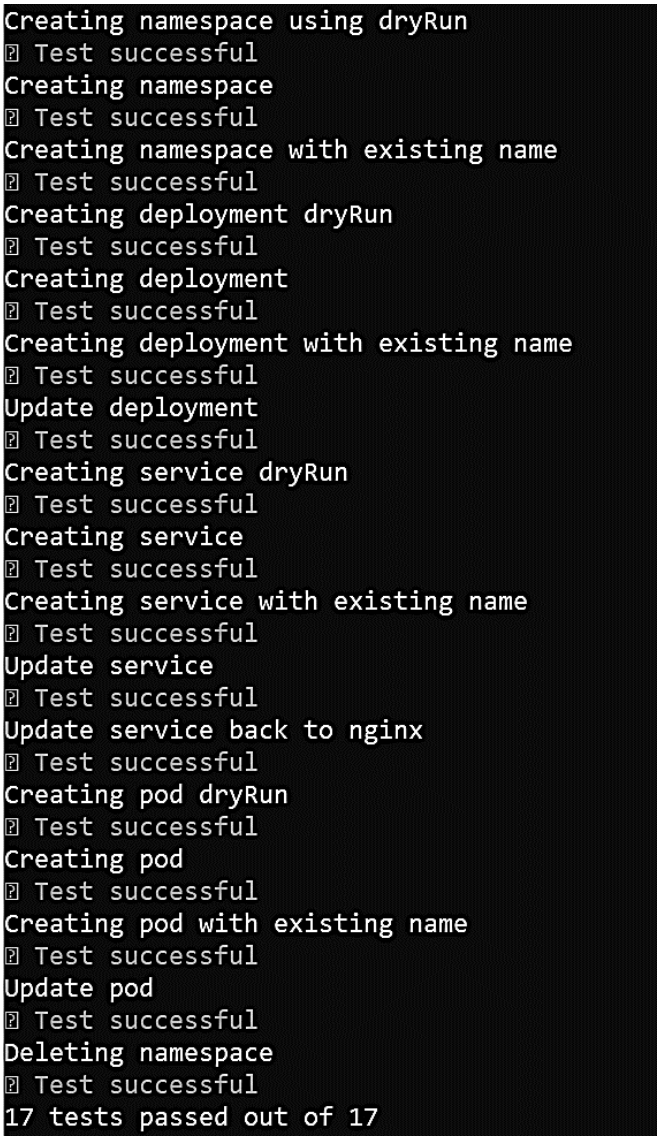

Figure 4 – POJO Model Action Unit Test Cases

An optional field called dryRun is added to each API call so that users can simulate CRUD operations on their POJO Model to determine the effects an operation may have on their Kubernetes cloud systems. Another route for differences is also provided so that users can generate differences between the model and cloud system to compare their development over time.

5.4. Variable Fetcher SpringBoot Server with Concurrency

The variable fetcher is designed to gather information from user-defined sources. As such, the variable fetcher must have modular components, also it must allow for concurrent requests to allow multiple variables to be fetched at once. Requests to the variable fetcher are routed to the respective URL classes and the information gathering components (Gatherer) are responsible for calling the appropriate function or API to create a variable. The information gathering

component returns a CompletableFuture to the URL class to allow for concurrency in requests. The response from the CompletableFuture is then asynchronously returned in a response to the user request. The basic structure of the variable fetcher is shown in Figure 5.

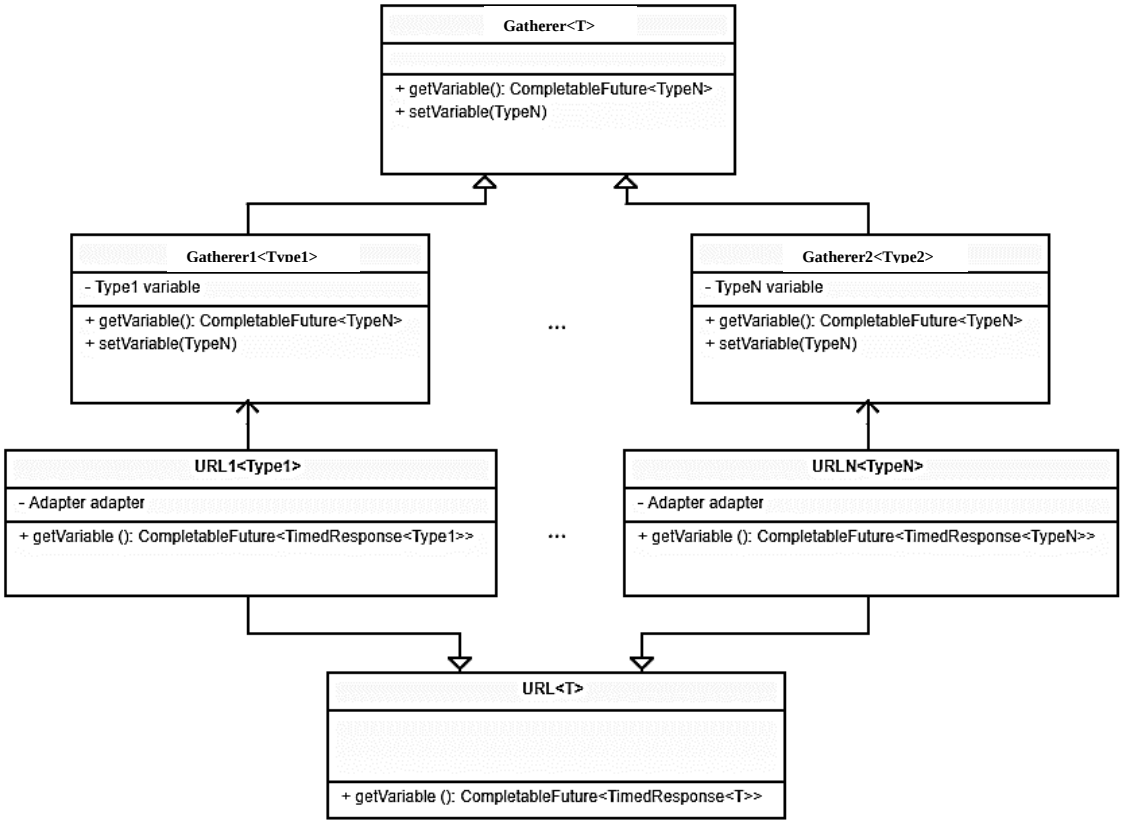

Figure 5 – Components of the Variable Fetcher

The variable fetcher test cases are shown in Figure 6 with three main object types: Integer, Double, and String. The Metrics test case was used as a special case to gather the CPU usage percent of a specific pod on a specific node as an integer.

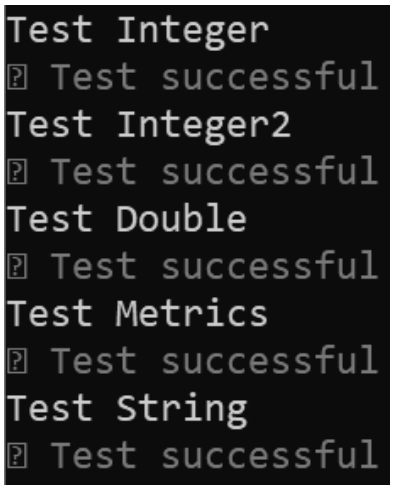

Figure 6 – Variable Fetcher Unit Test Cases for Different Object Types

5.5. Policy-Action Spark Server

The final component of the system was a policy-action server that would evaluate PolicyPlan objects every set number of time units. The policies would be evaluated using a set of variables that were gathered from the variable fetcher. While a PolicyPlan is satisfied the ActionPlan is foregone, however, if the policy is violated then the related ActionPlan is enacted.

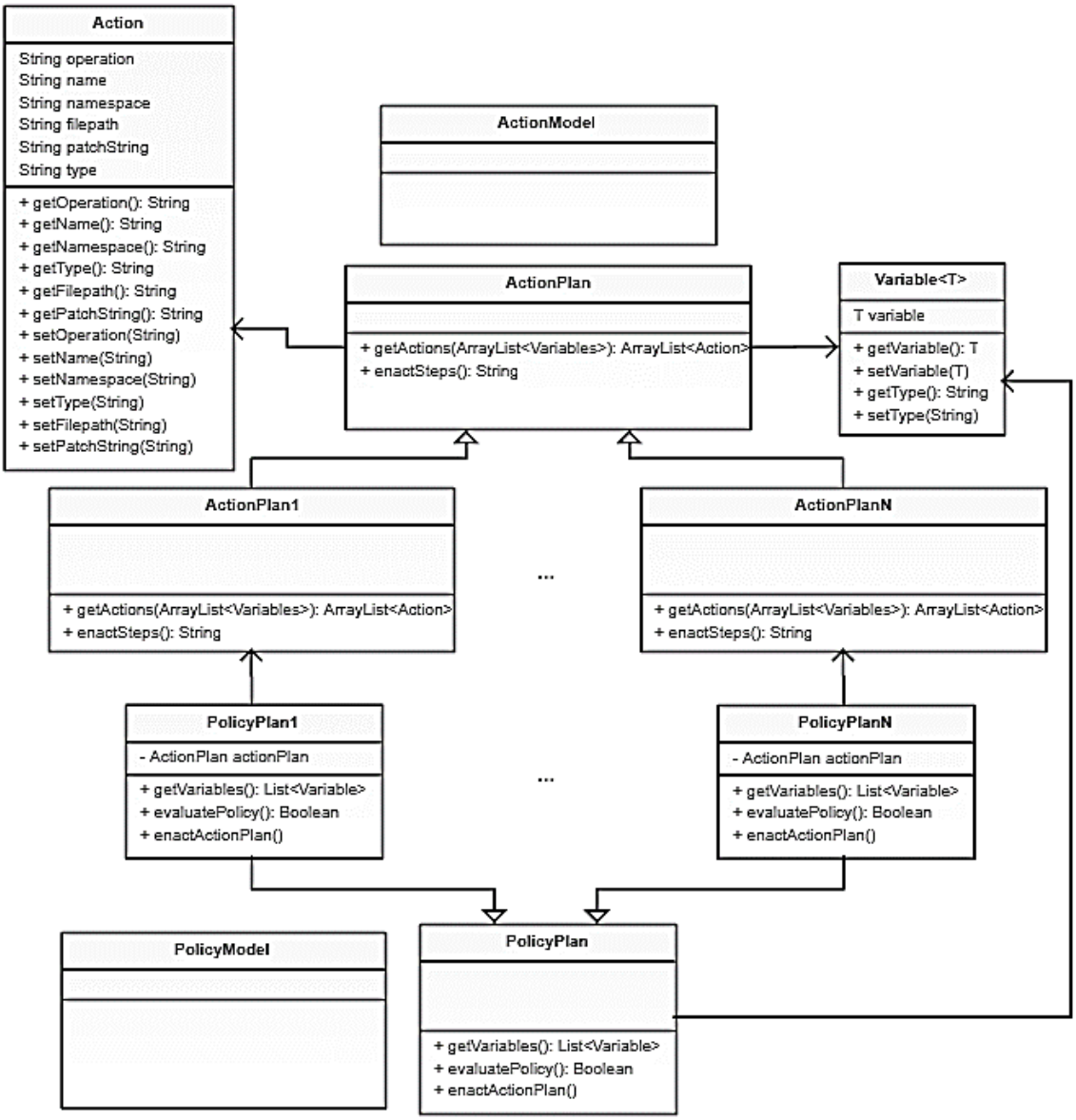

Figure 7 – Components of the Policy Action System. The ActionModel and PolicyModel are currently in development.

Once the policy-action server was developed integration testing using policies was performed. Figure 8 displays the results of this integration testing.

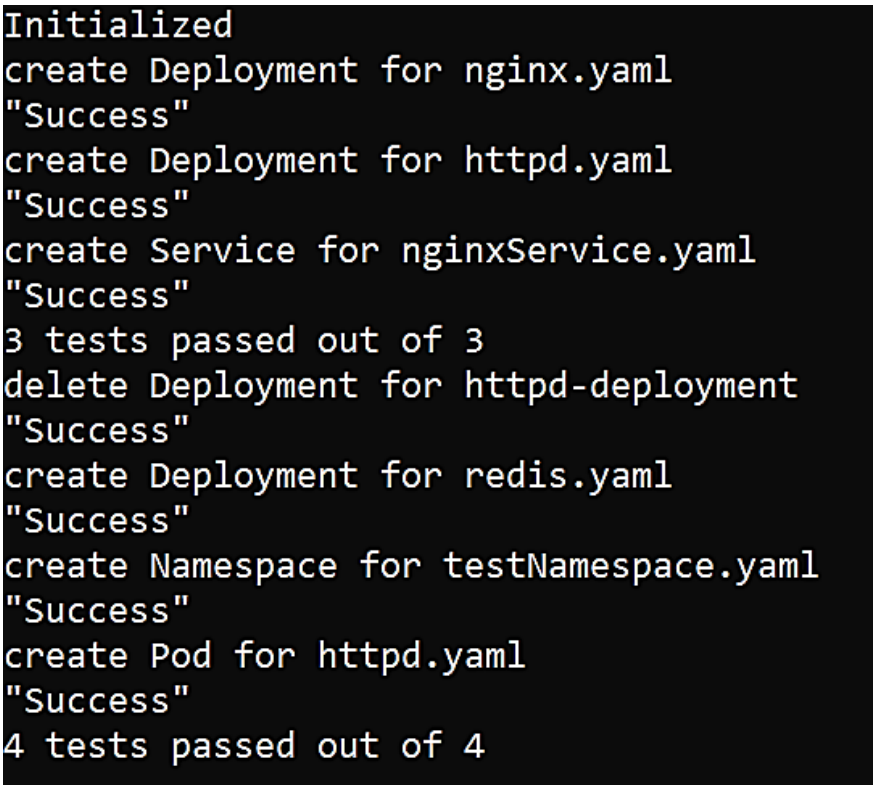

Figure 8 – Policy-Action Integration Test Cases

The integration test cases, in Figure 8, describe the case studies that were used to validate the system. The first integration test case is based on pod interdependency where the policy gathers variables related to pod information on the cluster and determines if a new pod should be created based on its' interdependency with another pod. For the second test, a metrics-based approach was taken, a Kubernetes-based metric was gathered as a variable and used in a policy as an integer. The final use case gathers variables from an external system like Jenkins to test a policy and enact reconfiguration actions. These are all important validation tests that demonstrate the wide applications of this system. Additionally, business related use cases such as rolling upgrades and different subscription tiers could be managed by such a dynamic reconfiguration system.

6. Discussion

6.1. Threats to the validity of the results

- The Kubernetes Java Client lacks relationship information because it does not use association, composition, or aggregation for its relationships between resource classes
    - Solution: Client APIs were used to gather the missing relationship information
- If CRUD operations on resources manipulate other child resources, these manipulations are delayed and the child POJOs cannot be collected immediately
    - Solution: Add a wait, or delay to the API calls

6.2. Implications of Research Results

- Benefits to background and related work:
    - System can be used to more completely model Kubernetes systems because it is directly linked with the Java Client API developed by the Kubernetes team
    - A more generalized approach is used by this system to allow programmers the freedom to create their own dynamic reconfiguration pipelines
- A combined MDD and policy-action cloud reconfiguration system could be used to reduce overall costs and increase efficiency of development cycles [2]
    - Manual reconfigurations involve a significant amount of overhead
    - MDD is used to generate abstracted model-based procedures that can reduce accidental redundant complexities of software development [4]

6.3. Limitation of the Results

- Only four of the thirty-one resource types have CRUD operations because the most commonly used resources were sampled
    - Developing CRUD operations for all the resources using the Kubernetes Java Client API would remove this limitation
- The POJO Model is stored statically in a Java singleton class, this is not a persistent way to maintain a Model
    - Store the POJO Model is a database to maintain its' persistence in case the server suffers from interruptions

6.4. Generalizability of the Results

- The results are highly generalizable because the system is made to be generic so that users can generate their own model, policies, actions and variables
- Additionally, some of these policies have potential uses in production because they can limit specific applications from users while sharing other applications depending on the user's subscription

7. Conclusions

In literature current metamodels for Kubernetes systems are limited by the lack of detail to appropriately describe the Kubernetes resources and their interactions. Although there are strong MSA-MDD approaches to central-architecture problems, as demonstrated by the MDD of AWS, there is an underrepresentation of MDD-MSA approaches to service development across teams [4].

To create an MDD-MSA approach for dynamic reconfiguration, three main components were developed: a POJO Model Spark server, a policy-action Spark server, and a Variable Fetcher SpringBoot Server with Concurrency. The POJO Model server consisted of a POJOModelService singleton class used to hold the state of the model. Also, a POJOModelController class to route the CRUD operations for Deployments, Pods, Services, and Namespace to one of their CRUD operation classes. A variable fetcher SpringBoot server was

used to gather any system, service or external variables that were needed to evaluate a policy with support for concurrent API calls. These variables were served on their own individual routes to be called during policy evaluation, the URLs as adapters were mapped to Gatherer classes as adaptees. The last component was the policy-action Spark server which integrated the POJO Model and variable fetcher components to evaluate a policy plan as an adapter to an action plan as an adaptee.

In this paper, a dynamic policy-action system is combined with a POJO Model that is based on Kubernetes Java Client to develop a novel approach for dynamic reconfiguration of Kubernetes systems. In the research field, there is a general lack of application-based studies that demonstrate the principle of dynamic reconfiguration. As a result, it is crucial to generate and study such systems to produce more cases of their usage. The case studies covered in this project include cases where local variables, metrics variable, and external variables are used to enact policy action triggers.

The new approach described in this research is beneficial to both development and production environments. In development, the dynamic reconfiguration system will foster communication between teams as they will have to work with a centralized model to craft their microservices. Additionally, instead of supporting scripting of dynamic reconfiguration, this framework helps to separate the different facets of the process so that users can build the facets separately, reducing inter-dependency and allowing for modularization and abstraction of microservice orchestration pipelines. As for production environments, such a system could be used to reduce overall costs and increase efficiency [2]. New updates could be rolled out one at a time allowing users to have the newest versions of specific microservices.

8. Future Work and Lessons Learnt

8.1. Future Directions

- Add the CRUD operations for the remaining 26 resources
- Create a Plug-And-Play Architecture for the policy-action and variable fetcher servers
- Store model in a persistent storage because it is currently stored in
- Tree decision models that involved tree traversals to identify the appropriate policies and actions to test for each PolicyPlan and ActionPlan
- Look into Fabric8's implementation of the Kubernetes Java Client because it is a popular alternative to the native Kubernetes Java Client

8.2. Lessons Learnt

- There has been a recent rise of MDD with new areas of research
- Sometimes the easier option is more effective, because using the existing Kubernetes Java Client was much easier than developing a completely new Ecore Model

9. Acknowledgements

I would like to thank Dr. Konstantino Kontogiannis, Dr. Nazim Madhavji, Kostas Tsiounis (PhD Student) and Jorge Fernandez (Undergraduate Research Student) for all their support. It would not have been possible to achieve this project without their help. Also, I would like to thank Western University for giving me this opportunity.

REFERENCES

[1] J. Kramer and J. Magee, “Analysing dynamic change in distributed software architectures,” *IEE Proceedings - Software*, vol. 145, no. 5, p. 146, 1998.

[2] A. Verma, G. Kumar, and R. Koller, “The cost of reconfiguration in a cloud,” *Proceedings of the 11th International Middleware Conference Industrial track on - Middleware Industrial Track 10*, 2010.

[3] A. Evangelidis, D. Parker, and R. Bahsoon, “Performance Modelling and Verification of Cloud-Based Auto-Scaling Policies,” *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2017.

[4] J. Sorgalla, F. Rademacher, S. Sachweh, and A. Zündorf, “On Collaborative Model-Driven Development of Microservices,” *Software Technologies: Applications and Foundations Lecture Notes in Computer Science*, pp. 596–603, 2018.

[5] M. Zúñiga-Prieto, J. González-Huerta, E. Insfran, and S. Abrahão, “Dynamic reconfiguration of cloud application architectures,” *Software: Practice and Experience*, vol. 48, no. 2, pp. 327–344, 2016.

[6] T. F. Düllmann and A. V. Hoorn, “Model-driven Generation of Microservice Architectures for Benchmarking Performance and Resilience Engineering Approaches,” *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion - ICPE 17 Companion*, 2017.

[7] M. Zúñiga-Prieto, E. Insfran, S. Abrahão, and C. Cano-Genoves, “Automation of the Incremental Integration of Microservices Architectures,” *Complexity in Information Systems Development Lecture Notes in Information Systems and Organisation*, pp. 51–68, 2017.

[8] C.-T. Lee and R. B. Williams, “Dynamic detection and reconfiguration of a multi-tenant service,” 10-Apr-2018.

[9] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. New Dehli: Pearson Education, 2015.