DEPO: A Platform for Safe DEployment of POlicy in a Software Defined Infrastructure

Aisha Syed
University of Utah
aisha.syed@utah.edu

Bilal Anwer
AT&T Research
bilal@research.att.com

Vijay Gopalakrishnan
AT&T Research

Jacobus Van der Merwe
University of Utah

ABSTRACT

The emergence of network functions virtualization (NFV) and software defined networking (SDN) has resulted in networks being realized as software defined infrastructures (SDIs). The dynamicity and flexibility offered by SDIs introduces new challenges in ensuring that policy changes do not result in unintended consequences. These can range from the breakdown of basic network invariants to degradation of network performance. We present the Depo framework that enables automated discovery and quantification of the potential impact of new orchestration and service level SDI policies. Our approach uses a combination of knowledge modeling, data analysis, machine learning, and emulation techniques in a sandbox SDI. We demonstrate our approach by evaluating it over a testbed SDI with a 4G LTE/EPC broadband service.

CCS CONCEPTS

• **Networks** → **Network management**; **Programmable networks**.

1 INTRODUCTION

Emerging software defined infrastructures (SDIs), consisting of software defined networking (SDN), network functions virtualization (NFV), cloud computing, etc. are being embraced by network service providers and equipment vendors [5, 32, 44, 48]. The inherent flexibility and agility of an SDI environment is enabling better resource management [25], rapid service composition [46, 65] with virtualized network functions (VNF) [4, 16, 39, 47], dynamic service deployment and evolution [21, 61], custom data planes [10, 13, 26, 49], and control plane modifications [8, 63]. SDIs are anticipated to be key enablers for future mobile network architectures and related services being developed for 5G [2, 3, 43, 44, 51, 56].

There is immense push in the industry and academia towards closed-loop, policy-driven control platforms for the SDI ecosystem [1, 5, 14, 16, 17, 38, 46, 58, 61, 64]. The SDI control platforms being proposed consist of controllers and orchestrators (with an associated policy engine) that utilize network and service templates (specifications) to automate the workflows used for managing the physical and virtual infrastructure that creates the SDI, and for the composition, instantiation, evolution, and lifecycle management of the services that run on top of it. The way automation is expected to be done and change enacted in this ecosystem, is through policies deployed via the policy engine. Policies are essentially rules, conditions, requirements, constraints, attributes, or needs that must be provided, maintained, and/or enforced. At a lower level, policy involves machine-readable rules enabling actions to be taken based on triggers or requests. Policies often consider specific conditions or events, i.e., triggering specific policies when conditions are met, and/or selecting specific outcomes of the evaluated policies appropriate to current conditions [46]. This benefit of policies in allowing rapid modification and updating of behavior without rewriting software code fits very well in the complex and dynamic SDI environment.

Earlier policy related efforts mostly focused on low-level configuration and networking policies (e.g., BGP or SDN rules), with work being done on controlling and verifying their impact [19, 27, 28, 38, 42, 68]. The SDI environment, however, has a need for abstractions that allow writing orchestration and service level policies in addition to the traditional low-level networking policies and configuration updates. Since SDI control platforms are being designed to be templates-based, the domain experts can thus write policies in high-level abstractions using references to mechanisms specified through these SDI object templates [61, 64] (possibly created by different domain experts and vendors). Moreover, the SDI is inherently layered (physical infrastructure, virtual infrastructure or VNFs, and the service layers), with complex horizontal and vertical

SOSR '19, April 3–4, 2019, San Jose, CA, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6710-3/19/04…$15.00
https://doi.org/10.1145/3314148.3314358

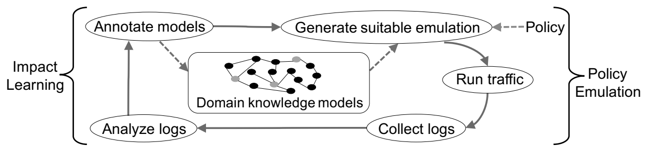

Figure 1: Depo: Continuous learning

interactions between objects at the various layers, controlled by policies from potentially independent actors. This cross layer interaction and presence of multiple actors coupled with the dynamism due to capabilities from NFV/SDN (scaling, live migrations, live topological updates or insertion of new functionality [21, 61]) makes it non-trivial and impractical for humans to manually trace the impact of their policies.

Thus, there is a need for an automated tool to be coupled with the SDI such that SDI policies can be tested for impact before being deployed in production. To this end, we present Depo. Depo is an end-to-end framework for checking the impact of policies in an SDI environment. As shown in Figure 1, Depo represents a continuous learning approach whereby a knowledge based model captures the relationships between objects in the SDI as well as the impact of policies on them. The Depo learning process involves emulations under varying conditions, monitoring and data collection of the emulated environment, statistical and machine learning based analysis of the data to enable discovery and quantification of impact relationships between policies and SDI objects. This continuous learning approach enables Depo to continuously annotate the knowledge base as data is collected through emulations in varying environments, and the confidence level in the learnt impact relationships increases. Depo uses a greedy approach of testing the policy in emulation environments that show potential for resulting in more information gain, as determined from data analysis from past emulation runs. The knowledge-based approach allows inferencing so that Depo can find not only the SDI objects that were significantly impacted by the given policy, but can also infer how the affected objects are related to the object on which the policy was deployed. This capability helps in determining both cross-layer and within-same-layer relationships where the policy writer may not have knowledge about *how* their policy impacted other SDI objects.

We make the following main contributions.

• We propose the Depo framework which, given SDI orchestration and service-level policies, allows us to determine and quantify the impact of such policies on objects in the SDI.
• We present a systematic approach to structuring domain knowledge in a model to enable more informed policy writing and policy impact checking. We further annotate and verify the initial domain knowledge model by incorporating knowledge learnt through SDI emulations.
• We use statistical analysis and machine learning (ML) to enable knowledge-based inference and reasoning to further describe *how* a given SDI policy affects the SDI objects.
• We present a prototype implementation of Depo and its evaluation using example policies in a testbed SDI environment. We show Depo is capable of capturing the impact of policies and can reuse and revise learnt knowledge across emulations.

2 RELATED WORK

Traditional approaches to enacting network change include multiple stages during which network changes can be checked and validated before deployment, e.g., manual planning, static or online verification and validation [18, 27–29, 35, 52, 59], simulation and emulations [34], phased rollout or first field application (FFA) [37], and canaries [69]. The focus of these efforts has either been on low-level configuration updates related to the network and routing layers, or has enabled limited automation and required the presence of domain experts for designing the test environment needed for policy or update checking. With Depo, we consider the virtual and service layers made possible by the SDI and perform knowledge-based modeling to capture cross-layer interactions as well as automate the process of suitable test environment generation.

More related to our work are dependency checking and analysis [11, 36, 37], statistical and ML based analysis approaches [15, 22, 42, 62]. These earlier efforts create siloed models for specific use-cases, services, or network objects. In contrast, Depo's modeling approach makes it service-agnostic so that policies for any SDI services can be tested. Depo also has an analysis process that systematically bootstraps its knowledge base using service and network object specifications that have been presented and tested by the community for use in the SDI [61, 64]. Further, Depo performs automated emulation generations and tests the policies in various configurations, using a learning process that is continuous, i.e., learning across emulation iterations and improving learnt knowledge about policy impact over time. Finally, Depo's modeling and learning approach also allows learning cross-layer interactions and relationships which arise in the SDI ecosystem.

Our work is aligned with industry and academic efforts on policy-driven SDI systems that consider the definition of policies to be more than traditional networking and routing policies [1, 5, 14, 16, 17, 46, 61]. However, to the best of our knowledge, our work is the first to present SDI service and orchestration policies and develop a generic testing framework for learning their impact in the SDI ecosystem. Depo thus serves as a practical first step towards exploring the challenges and opportunities in this space through prototyping an end-to-end impact discovery/checking system.

There is ongoing research in the ML community for building tools that can help humans interpret ML models [6, 30, 31, 33, 54, 55]. This is complementary to our work since one of our goals is to encode the learnt policy impact so that it is both machine and human interpretable, i.e., actionable knowledge that the machine can utilize in the continuous learning process, and humans can utilize in understanding their policies' impacts. However, in contrast to Depo, these efforts output model interpretations that are still at a lower level.

3 CONTEXT AND CHALLENGES

3.1 Depo Context and Use-Cases

SDI Services. We take the standard 4G LTE/EPC broadband [45] as our first use-case service. Figure 2a shows the high level EPC architecture consisting of the Radio Access Network (RAN) and the Evolved Packet Core (EPC). eNodeBs (base stations) are part of the RAN and wirelessly connect to the user equipment (UE) i.e., mobile devices. We assume a software-defined core and RAN [51]. The EPC core has three main components. The MME (Mobility Management Entity) is a control plane entity and handles UE authentication, registeration, and mobility. The SGW (Serving Gateway) is a datapath element and forwards user traffic coming from the RAN to the PGW (Packet Data Network Gateway) which serves as the gateway to external networks. The S/PGW and eNodeB also handle control plane functions.

Figure 2b shows an EPC variant with selective edge cloud traffic offloading functionality. An implementation of this architecture is SMORE [12] and we take it as our second use-case SDI service. We use webservers as the example low-latency apps in the edge cloud.

Terminology. Here we define terminology used in the paper. In the SDI ecosystem, parameterizable *templates* [61, 64] serve as the specification or 'object type' for SDI components and services. Instantiating them through an SDI orchestrator causes *instances* or *objects* of them to be created in the SDI. This is similar to classes and objects (instances) in an object oriented programming language. Templates can be 'container'-type templates such that they describe a type composed of one or more other SDI object types. E.g., a *service* template, such as, EPC, can be composed of component templates, such as, MME, SGW, PGW, eNodeB. Similarly, a SMORE service template can be composed of components, such as, the Webserver in the edge cloud, any load balancer or cache components, etc.

Templates contain variables, lifecycle and management mechanisms, and policies. Mechanisms can be related to routing (push/remove/update flows), scaling, load balancing, migration, performance tuning, etc. For example, init, start, stop, migrate, scaleUp, update, configureIPv4, are some typical mechanisms available in SDI component or service templates.

Table 1 shows example variables in the templates of various SDI object types we considered. For space reasons, some of the related variables are shown as grouped together, e.g., Server.ifaceVars and VNF.migrationVars represent variables related to Server interface configuration (e.g., IPv4Addr, MACAddr) and VNF migrations (e.g., numMigrationsPast5Min, or

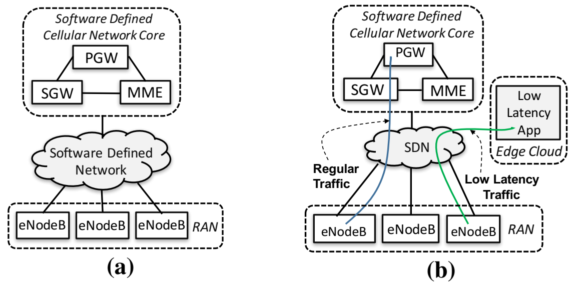

Figure 2: (a) EPC, (b) EPC with edge cloud offload

numFailedMigrations). Also, variables from different VNFs (e.g., S/PGW, SMORE_Webserver or SMORE_Cache), and Services (e.g., EPC, SMORE) are shown grouped together under the VNF and Service headers.

For systematic impact analysis, we categorize variables as: (1) *configuration/configurable*, (2) *observed*, and (3) *workload* variables. In addition, the domain expert or policy writer can tag certain configuration and workload variables to be (4) *emulation environment parameters*. We assume a domain expert annotates them as such in the templates.

Configuration variables are template variables that can be modified (or *configured*) directly using template mechanisms, e.g., IP address, location (configurable using e.g., VNF migrations), num of VMs, cpuOversubscription, num of CPUs, etc. In addition, we also consider configurable *emulation environment parameters* to be part of this category. E.g., Server's num of CPUs is considered a configurable variable since it can be varied across emulation runs by picking Server types that have varying number of CPUs. We define *observed variables* or *(performance) metrics* as those that cannot be directly configured using mechanisms or emulation parameter selection. E.g., CPU usage, average latency seen by subscribers in a location, percentage of failed UE handoffs, etc. Finally, *workload variables* represent features of service workloads e.g., num of UEs, request rates, etc. Values for these are encoded by the domain expert or policy writer running the emulations. E.g., their values can come from realistic traffic datasets, or the policy writer may want to also use values that have not been seen in realistic datasets. These can also be tagged as *emulation environment parameters* in which case they are used to control workload generation during emulation runs for

Table 1: Example variables in the SDI

Server	**VM**	**Switch/Link**	**VNF**	**Service**
location	location	location	location	topologyVars
status	status	status	status	numENB
rateStatusChange	rateStatusChange	ifaceVariables	type	numSGW
totalMem	totalMem	rateStatusChange	totalMem	numPGW
allocatedMem	numCPU	numFlaps	numCPU	numMME
numCPU	version	latency	version	numWebserver
numAllocatedCPU	type	totalMem	cpuUsage	latency, throughput
cpuOversubscription	cpuUsage	numCPU	memUsage	...
numAllocatedVM	memUsage	version	throughput	**Workload**
numRunningVM	migrationVars	type	latency	numUE
version, memUsage	numNeighborVM	cpuUsage	topologyVars	rateOfRequests
cpuUsage, type	osImage	memUsage	migrationVars	interarrivalTime
ifaceVars	ifaceVars	propagationDelay	cacheVars	num/rateOfMobility
...	...	...	...	concurrentVNF/Service

policy impact checking under various workload conditions. Since these are not relevant to template instantiation they are typically not captured as part the of SDI templates. However, we extend the templates to add them.

The goal of Depo is determining the impact of policies; we define 'impact' systematically as follows. Policies when activated, invoke object mechanisms, which in turn *affect* or *impact* configuration variables (i.e., modify their values), which in turn *affect* the observed variables (or performance metrics), in the presence of workload variables (e.g., request rates). We capture this entire trace, from policy, down to the configuration and observed variables impacted. This is because both configuration and observed variable impacts can be due to unintended consequences which were not part of the policy writer's intent. We consider a statistically significant change in the value of a variable as an impact on that variable. Significance level is a tunable parameter (e.g., 95% confidence using p-values with alpha set to 0.05). When a policy is tested, the correlated changes at the given significance level, on object variables in the context of an object template are output as impact.

Policy examples. Listings 1 to 6 show the pseudocode for example policies used in the paper, written for our SDI environment with the two services introduced earlier: EPC, and a variation of EPC with edge cloud offloading (referred to as SMORE in the paper). We consider if-then style policies that are written in the context of mechanisms and variables exposed by SDI object templates.

The policies are self-explanatory. Listing 1 shows a Server update policy which applies available updates on all servers in the edge cloud location. Listing 2 shows a Server policy that enables CPU oversubscription by some threshold if the CPU usage has been high i.e., it causes the SDI orchestrator to pack more VMs on the Server instances by oversubscribing the CPU resource. Listing 3 shows an EPC service policy for SGW scale up. Listing 4 shows an SMORE service policy for scale up of its webserver component. Listing 5 shows a SMORE policy that enables caching for the webservers if SMORE service's subscribers are seeing a higher latency. Finally, Listing 6 shows a cross-service policy where if some EPC service's subscribers are seeing a higher latency to an Internet webserver, then the SMORE service is dynamically enabled for specific subscriber UEs, i.e., selective edge cloud offloading is enabled for this EPC instance for subscribers mentioned in the subscriber list. This would include the SDI orchestrator instantiating SMORE service components (e.g., webserver in the edge cloud), and configuring networking (e.g., SDN rules) so that EPC traffic for specific subscribers is diverted to the webserver in the edge cloud.

As seen from the listings, we allow the policy writer to specify policy variables, such as, threshold variables, and the associated range of values they can take. E.g., for Listing 2, the writer can specify the range taken by THRESH2 to be [10, 50, 100] to test only few oversubscription percentages, or [1 .. 100] to test percentage values ranging from 1 to 100. Since the SDI is a very dynamic environment, hardcoding threshold values does not always result in intended consequences. Thus, during emulations, Depo iterates on the given policy variables (through a 'knob turning' process described later) and can output the learnt impact seen for them.

Listing 1: Update server

```
when:
   Server.updateAvailable == True AND
   Server.locatedAtEdge == True
then:
   Server.update()
```

Listing 2: Oversubscribe

```
when:
   Server.cpuUsage_10minAvg < THRESH1
   //THRESH1 = 50%
then:
   Server.setCPU_Oversub(oversubPerc =
          THRESH2) //THRESH2 = 50%
```

Listing 3: Scaling SGW

```
when:
   SGW.cpuUsage >= THRESH
   //THRESH = 70%
then:
   EPC.scaleUpSGW()
```

Listing 4: Scaling SMORE

```
when:
   SMORE_Webserver.cpuUsage_5minAvg
               >= THRESH  //THRESH = 70%
then:
   SMORE.scaleUpWebserver()
```

Listing 5: SMORE caching

```
when:
   SMORE.subscriberLatency_5minAvg >=
         THRESH
   //THRESH = 30ms
then:
   SMORE.setCaching()
```

Listing 6: SMORE offload

```
when:
   EPC.subscriberLatency_10minAvg >=
         THRESH
   //THRESH = 30ms
then:
   EPC.augmentSMORE(subscriberList)
```

3.2 Challenges and Solutions Overview

• ***Large number of knobs.*** There is typically a large number of *knobs* or test environment variables that need to be varied when testing a policy for impact. These include e.g., emulation parameters and workload variables, as well as threshold variables specified in the policy. A large number of knobs can make the policy impact checking costly and it is also non-trivial to manually set up test environments and perform impact checking. In Depo, we deal with this by automating the test environment generation using emulations, and perform systematic impact checking using a greedy approach of *knob turning* which first varies and explores knobs that result in more information/knowledge gain in terms of their impact on observed variables or performance metrics of interest.

• ***Dynamic environment and stale knowledge.*** SDI reduces the time to market for introducing new (or updated) services and/or components which makes it imperative to automate the continuous learning of policies' impacts. Traditional approaches and related work in policy impact learning that manually create siloed models and enable one-time learning cannot work in this dynamic environment. Depo deals with this by systematically modeling domain knowledge coupled with using templates that together allow automation of emulation environment generations for testing the impact of policies. Depo continuously updates the knowledge base with each emulation, adding to or improving existing knowledge about impact relationships between SDI objects.

• ***Layered architecture in SDI*** with different actors publishing templates for object types related to their own domain. It is non-trivial for policy writers to go through details of every template and trace dependencies and impact flows. Also, policy evaluation is dynamic and dependencies may exist at runtime that were not easily extractable statically. E.g., SDI enables newer operations, such as, dynamic changes to service topology or architecture, that can make it non-trivial to figure out generic relationships manually. In DEPO, we use modeling using a knowledge based approach to deal with this and systematically determine dependencies through inferencing in the knowledge base. This inferencing also considers knowledge annotations or modifications that occur due to DEPO's continuous learning approach. We couple this dependency discovery with data analysis and ML based methods to find the flow of impact among the dependent objects.
• ***Non-trivial to quantify variable-level relationships.*** While domain experts may know dependency relationships between various object types, however, it is non-trivial to discover and quantify variable-level knowledge, e.g., how a Server's CPU oversubscription variable affects the hosted SGW's processing speed and in turn affects EPC service's response time. Again, this process is automated using our emulation framework and subsequent data analysis that generates ML models that help quantify variable-level impacts.

4 THE DEPO SYSTEM

4.1 Architecture Overview

The DEPO architecture is shown in Figure 3 in the context of a *Software Defined Infrastructure (SDI)* [1, 14, 46, 61]. The SDI is a template-driven environment where an *Orchestrator* receives service requests and makes use of *Templates* and a *Topology Database* to instantiate virtualized service instances on its managed infrastructure. DEPO interfaces with the SDI to run emulations and analyze the impact of policies on the managed infrastructure and the service instances it hosts. We assume that, for the purposes of emulation, DEPO has access to a *sandbox* SDI instead of interfacing directly with a *production* SDI (which is left as future work). This sandbox can be a lab network associated with the production SDI.

DEPO has two main components: the *Policy Stager* performs the process of policy impact learning, and the *Knowledge Base* records the learnt knowledge. The Knowledge Base consists of a *knowledge graph* (KG) [23, 50] data structure which is used to encode domain knowledge, and associated ML models created during the impact learning process. Section 4.2 discusses the knowledge modeling done by DEPO in detail while Section 4.3) describes the impact checking process of the Policy Stager.

A KG captures knowledge in the form of *facts* which are 3-tuples of the form `"entity1 relationship entity2"`, e.g.,

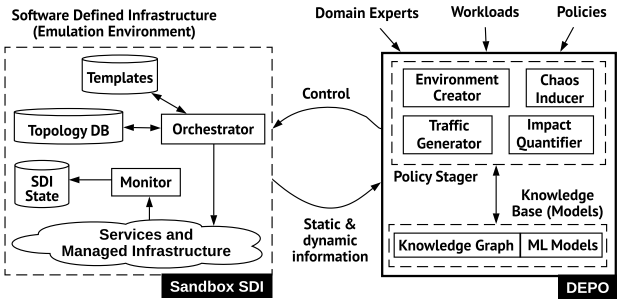

Figure 3: DEPO in the context of an SDI.

`server hasVariable serverCpuUsage`, or `serverCpuUsage affects VNFLatency`. KG is populated with both static and dynamic knowledge. *Static Domain Knowledge* is composed of knowledge provided by *Domain Experts* in the form of Knowledge Models that capture known relationships between entities at all layers in the infrastructure, as well as knowledge gleaned from Service and Topology information obtained from the Service Template Repository and the Topology DB in the SDI.

Dynamic knowledge includes knowledge learnt during the policy impact checking process performed by the *Policy Stager*. This involves the Stager using the SDI to create emulations, deploying policies, and annotating the KG with dynamic knowledge learnt through analyzing the information collected during these emulations. This information comes from traces of Orchestrator actions and templates' mechanism executions, and monitoring logs from the infrastructure. To enable this, the orchestrator and the templates are instrumented to log mechanism invocations (traces), and the policies themselves are instrumented as well. The interface to the SDI makes these logs available to the Policy Stager for analysis.

The *Environment Creator* in the *Policy Stager* takes policies as input, and generates suitable emulation environment configurations and directs the SDI orchestrator to instantiate them. It also generates configuration related to traffic workload variables and interfaces with the *Traffic Generator* to control traffic generation in emulations. This allows policy impact checking in varying workload conditions.

The *Chaos Inducer* is used by the *Policy Stager* to perform chaos engineering [9] during these emulations, e.g., killing servers and service component instances according to a given failure distribution, or cause performance degradations to create noise during emulation runs.

The *Impact Quantifier* analyzes the emulation logs and systematically quantifies the policy's impact, and in doing so, the impact of configuration variables and mechanisms available on object types that are referred to by these policies. The main goal of this learning process is to annotate the KG with learned knowledge about impact. If a generic relationship is observed between X and Y such that X is seen to 'affect' Y, then we create an 'X affects Y' relationship in the KG. Here, X and Y can be mechanisms, variables, policies. Due to the continuous learning process, the *Policy Stager* directs further emulation creations using the Environment Generator based on analyses from the Impact Quantifier.

4.2 Modeling Knowledge in KG

In this subsection, we describe the concept of a knowledge graph (KG), and show how we organize knowledge in a KG to facilitate and enable automated policy impact detection.

The unit of data storage and modeling in a KG is a *fact*, a 3-tuple of the form `(entity1 relationship entity2)`, where each entity is represented by a node in the KG and the relationship is shown using an annotated edge connecting the two related nodes. Multiple relationship edges with varying annotations can exist between the nodes which allows us to have different applications insert their own semantics over the raw data in the KG by adding overlays created using different relationship edges. Additionally, the nodes and relationships can be annotated with properties. Specifically, we add timestamps as properties, since we have the notion of time as part of our knowledge modeling. This allows KG to record changing state of the SDI and our query primitives can accordingly do time-based retrievals.

To enable generic inference, we use the concept of types and instances to create generic models representing an SDI environment. E.g., Figure 4 shows a high level view of such a model. A small part of the KG model we build for the virtualized EPC service is shown in Listing 7. The timestamp property is only shown on one of the facts for brevity. As the Listing shows, we encode generic model information from Figure 4 as static knowledge in the KG itself which allows our queries to be generic by only needing to understand the semantics from the model in the figure and then using that knowledge to query for specific instantiations of the model. This allows the query primitives built on top of the KG to not have any hardcoded domain knowledge about specific service instances embedded within them.

The generic knowledge itself can be at two levels, e.g., as seen from Figure 4, Service is a generic object or *type*. Then, EPC is an *instance* of that type and then EPC1 is a specific instance of EPC itself. The generic knowledge model at the first level is considered as static knowledge inserted by domain experts at system initialization time into the KG. Query and inferencing applications built on top of the KG need to understand the semantics of the model at this level. E.g., they may need to know what a 'hasComponent' relationship means, and how to use it to query for the components of any object of type Service. The generic knowledge at the second level does not need to be encoded within the query primitives but does need to be inserted into the KG. E.g., for an object of type Service, such as EPC, this involves taking knowledge about the generic service topology, control and

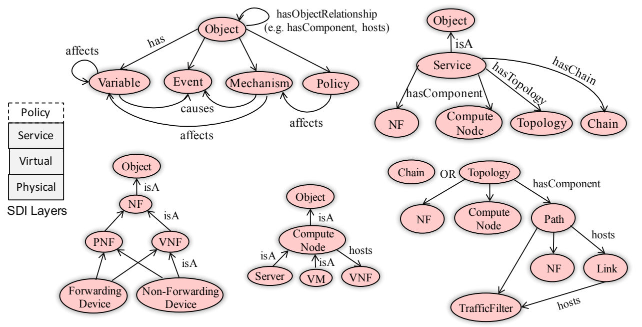

Figure 4: Example model

data protocol interactions, state and configuration variables, that exists in the heads of a domain expert, or in service and component templates in an SDI orchestration platform [60, 64], and inserting them into the KG as specific instance of the generic model. E.g., generic EPC relationships such as, EPC hasComponent SGW need to be encoded here.

Note that various experts and vendors can encode knowledge about their own templates or domains into the KG using the extensible modeling approach shown in Figure 4.

Once we have this static knowledge in the KG, monitoring applications can dynamically insert updates to the KG reflecting the current state of the SDI (i.e., service and network object instances like SGW1, or EPC1), and data-driven query primitives and corresponding impact detection applications can become possible. E.g., the orchestration of new service instances like EPC1, EPC2 can be logged by the monitoring module and inserted in the KG. This will cause new facts that represent the service instances to become available in the KG. E.g., facts about current protocol peerings, locations, performance metrics, and configurations. VNF migrations, load balancing, start and stop of components, configuration changes, and other changes in the state of the SDI will cause the KG to get updated accordingly.

A traditional *query* primitive in a KG is essentially a subgraph matching algorithm which takes as input one or more tuples or a subgraph in which one or more of the edges and nodes can be constant, acting as constraints, or they can be variables, and the query primitives then finds the bindings for those variables such that they follow the pattern dictated by the constants. For example, Listing 8 shows a KG query, where

Listing 7: Knowledge Graph

```
EPC isA Service
Service hasComponent NF
EPC hasComponent NF
EPC1 hasType EPC
EPC2 hasType EPC
VNF isA NF
MME isA VNF
MME1 hasType MME
EPC1 hasComponent MME1
Server isA ComputeNode
Server hosts VNF
Server1 isA Server
Server2 hosts MME1
MME.usage isA UsageVariable
MME1.usage hasValue 90% [timestamp=123]
```

Listing 8: Query

```
X_Var hasType EPC
X_Var hasComponent Y_Var
Return X_Var
```

X_Var and Y_Var are variables while the rest are constants. Running this query against the KG represented by Listing 7 will return the results [X_Var = EPC1; Y_Var = MME1] since only EPC1 matches this pattern and not EPC2.

4.3 Depo Process

Given the policy to check for impact, Depo Policy Stager learns its impact by emulating it in the SDI sandbox and learning from the collected logs. To be specific, it performs the following three tasks.

(1) ***Generating emulation environment*** suitable for testing this policy in the sandbox SDI.
(2) ***Running emulation in SDI sandbox*** by running traffic through it and collecting logs.
(3) ***Learning impact*** and annotating domain knowledge models by analyzing the collected logs.

As shown in Figure 5, these tasks can be divided into a total of eight steps. The rest of the section describes these steps.

4.3.1 Generating emulation environment.

1. Parsing policy. In order to run emulations for the policy, Depo first needs to generate an emulation environment which contains the objects referenced by the policy, e.g., an EPC policy will require instantiation of an EPC service in emulation. Static parsing allows Depo to extract the object types referenced in the policy specification. E.g., parsing the policy in Listing 3 will extract two object types: SGW and EPC. In the rest of the section, we will refer to this policy as the SGW scaling policy.

2. Extracting knowledge from KG. The statically extracted (parsed) object types aren't always sufficient to generate an emulation for the given policy. E.g., if EPC object type was extracted then each of its components will need to be instantiated on objects of type ComputeNode (e.g., Server or VM), however, these types were not explicitly specified in the policy. Moreover, if the SGW scaling policy were written in a way such that it only referenced a 'component' type such as SGW (and not EPC), then Depo would need to get the 'container' type for SGW, i.e., the EPC service type, so that it can instantiate EPC and all of its components on objects of type ComputeNode. This is because Depo cannot instantiate a detached component in the emulation, instead instantiation will need to happen at the service level. Availability of knowledge based models helps Depo automate the process of obtaining this information as follows.

For each parsed object from Step 1, Depo performs a 'radius' query that we implement on the KG to find a given object type's 'neighbors'. Radius is a conservative query. It considers logical/physical/protocol level neighbors. E.g., neighbors for the SGW type will include the Service it is a component of (encoded using *componentOf* relationships), SGW's protocol

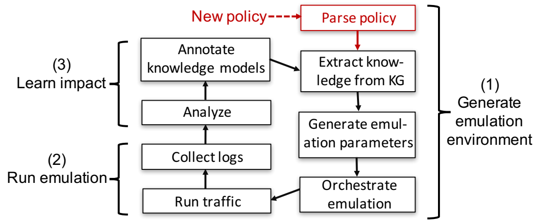
New policy
Parse policy
Extract knowledge from KG
Generate emulation parameters
Orchestrate emulation
Run traffic
Collect logs
Analyze
Annotate knowledge models
(1) Generate emulation environment
(2) Run emulation
(3) Learn impact

Figure 5: Policy stager process diagram.

neighbors (encoded using *hasNeighbor* relationships), and the type of objects that can host an SGW (encoded using *hosts* relationships). E.g., the KG will have facts such as, SGW isA VNF, ComputeNode hosts VNF, Server isA ComputeNode, Service hasComponent VNF, EPC isA Service, EPC hasComponent SGW, EPC hasComponent MME, etc. This knowledge comes from the modeling approach performed by domain experts as explained earlier in Section 4.2, and from extraction of knowledge from SDI templates. It is inserted into the KG at bootstrapping time (shown in Figure 3 using *static information* and *domain experts* labels).

Since generic knowledge related to neighbors (Figure 4) is encoded in radius query's implementation, this allows it to be generic and still be able to retrieve specific relationships about instances. E.g., radius understands generic concepts of Service, Component, and hasComponent relationships, and so knowledge about specific Service and Component instances such as EPC and its S/PGW components is extracted by radius automatically. Radius query thus allows Depo to get a list of the object types needed for emulation environment creation for the given policy. For the SGW scaling policy where SGW and EPC object types were statically parsed, Figure 6 shows the result of the radius search that extracts their neighbors in various directions. This forms the list of object types to be created in emulation.

By finding object types that are related to the policy and thus are under its impact radius, this Step 2 in the Depo process allows reduction in the number of object types that need to be created in emulation for the given policy.

3. Generating emulation parameters. Each object from the list of object types found at the end of Step 2 has configurable variables that are tagged by the domain expert as emulation environment parameters in the object's template. E.g., number of SGWs, and the number of its neighbor objects to create, the types of servers, and VM sizes to generate for hosting the SGWs and their neighbors, where VM.size, Server.type, EPC.numSGW, etc. are configurable parameters. Similarly, the workload that will be run on these emulation environments

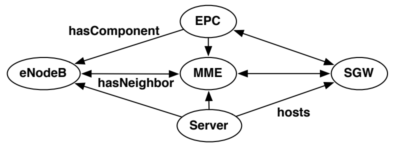

Figure 6: Update policy subgraph

also have workload variables that can be configured. E.g., rate of requests, or number of clients. Finally, the threshold variable THRESH can also be taken as an emulation parameter. We term all of these as knobs that need to be turned, where each configuration of the knob values can cause the policy to have a different impact on the SDI.

In Depo, we use a process of *knob turning* to try out the various knob configurations. By default, a new emulation is used for each *knob turn*. However, we assume that knobs that represent workload variables can be knob turned within the same emulation, e.g., traffic rate can be varied within the same emulation. Knob examples include configuration variables for object types, e.g., EPC.numSGW, EPC.numPGW, Server.type, VM.numCPU; and workload variables that can be used to control the workload generation process, e.g., EPC.requestRate, EPC.numClients, Server.otherVMs (for creating server load emulating other services and their VMs besides e.g., the one EPC instance for our current policy example).

In short, this Step 3 takes the list of objects output at the end of Step 2 and provides the list of knobs exposed by them to the knob turning algorithm (described in Section 4.3.4).

4. Orchestrating emulation. The knob turning algorithm iterates on the range of values for each knob and outputs a configuration of knob values. In this way, multiple knob configurations are output by the algorithm, where each configuration is then used by the Policy Stager to direct the orchestration of emulation in the sandbox SDI.

4.3.2 *Running emulation in SDI sandbox.*

5. Running traffic. The Policy Stager configures the Traffic Generator with the workload variable values output by the knob turning. The traffic generator is configured to generate traffic for various types of services present in the emulation. The duration for the emulations is a tunable parameter.

6. Logs collection. Traffic generation causes the emulation to generate logs. E.g., the logs collected for analysis include: information about network and service object instances and the logs for their state variables, and object instances' mechanism invocations (traces of policy action invocations).

4.3.3 *Learning impact.*

7. Logs analysis for impact checking. The analysis of these logs gives us the specific object IDs (instances) whose mechanisms were invoked by the policy and the state variable logs of those objects. This list of objects is under the policy's direct impact radius.

Next, for each object ID in this list, we do a 'radius' query in the KG which essentially traverses the neighbor relationships this object has with other objects. This helps us get a list of additional potentially impacted objects since they are neighbors to the directly impacted objects we had found earlier. We combine these two lists to create the (potentially) impacted objects list. Thus, this list creation process filters out all other object instances in the SDI that are not under the policy's impact radius.

A policy's impact travels from *policy action → mechanisms → configurable variables → observed variables*, in the presence of *emulation parameters* and *workload variables*. The goal of Depo's impact quantification is to find the presence of this impact and attempt to quantify it by learning from the collected logs and creating new models that capture the impact (or updating them if they already exist, with new logs). Multiple policies can use the same mechanisms and so it is useful to determine and record the relationship between mechanisms and configurable variables, and between configurable and observed variables.

For a given policy's action, first, the Impact Quantifier retrieves the during-action-execution, before-action-execution, and after-action-execution logs. How far into the past to get the logs from is controlled by TBefore and how far into the future to get the logs from is controlled by TAfter. Both are tunable parameters and can be set globally or per variable. The Quantifier performs before-during and before-after comparison of all variables that exist on objects in the object list from earlier. A 2-sample Kolmogorov-Smirnov (KS) test [40] is used for this comparison. Given two distributions, KS returns a difference statistic D, and a p-value. The 'alpha' value for KS is a tunable parameter. If the output p-value is less than or equal to the alpha value then the two distributions are considered significantly different.

Thus, the output of the KS tests gives us a table of mappings, $\{variable : CHANGE\}$, for all variables on the objects in our list, where CHANGE is a boolean variable with 'yes' if the KS test found that it was impacted by the policy, and 'no' otherwise. Note that, we combine the results for similar objects e.g., all object instances of type Server, thus larger number of instances depicting a large scale network in the emulation increases accuracy of the KS test since more data samples are generated. The KS test reduces the number of variables under consideration. Next, the Quantifier creates a list with all variables that have CHANGE=yes in the table. The list is further divided into two lists, for configurable variables, and observed variables.

Supervised regression or classification machine learning models can now be created where the configurable variables are taken as features, and their specific effect is observed on the observed variables of interest (e.g., latency, usage). Depo takes a pluggable list of regression and classification model implementations and selects one of them automatically using a grid search technique that decides which one to select depending on whether the data is continuous (regression selected) or categorical (classification), and the resulting accuracy of the model. During the search, the dataset is divided into training and test sets with specific percentages (e.g., 20% as test data, 80% as training). k-fold cross validation [66] is used to further test for bias and produce model accuracy values, the model with the highest accuracy value is chosen.

While the earlier KS tests tell us that there is 'an impact' or 'an effect' between the configurable and observed variables (a Yes/No answer), the machine learning models enable more fine-grained impact finding and capture knowledge about the type of relationships (e.g., linear, non-linear, exponential, etc.), and the environment in which these relationships apply (e.g., other environment variables captured as features in the models). Note that one model is created for each observed variable under consideration (where CHANGE=yes was observed), where the configurable variables serve as features, and the observed variable as the dependent variable.

8. Annotating domain knowledge models. The models and probabilities learnt are recorded for later querying by policy writers, and are continually updated when more logs get collected through new emulations, multiple policy triggerings in time within the same emulation, or a policy triggering in the same emulation but on multiple similarly typed objects (object instantiations of the same template).

The variables that were tagged as having CHANGE=yes earlier are recorded within the KG as 'affects' relationships among variables, mechanisms, and policies. These 'affects' relationship edges are annotated with the p-values obtained using KS tests mentioned earlier. We use Fisher's method [20, 24] of combining p-values when we update existing facts with new results. This combining process depicts increase or decrease in the strength of the affects relationships. That is, due to these updates, if the p-value on an affects relationship becomes less than or equal to a given alpha value (e.g., common alpha is 0.05 for getting 95% confidence) then it depicts the KG nodes connected by the affects relationship have a statistically significant affect on each other (e.g., with 95% confidence if alpha is set to 0.05). Thus over time, Depo allows continuous learning by either adding new affects edges or by updating existing ones and thus helps in improving the knowledge model over time.

Depo allows domain experts to encode affects relationships between variables based on their past knowledge and assumptions at bootstrapping time of the knowledge model. However, these may not always be correct. The continuous learning capability of Depo helps here by allowing automatic correction of this knowledge using new logs collected through emulation runs. Thus Depo can correct existing affects relationships or create new ones if they are missing in the knowledge model but are learnt through emulations. Note that while Depo can deal with such incorrect or missing knowledge, however, it cannot automatically deal with missing object types or variables. Depo assumes that static knowledge obtained from templates is complete and correct, e.g., knowledge about components of a service, the configurable, observed, and workload variables, and the available object mechanisms. This is a reasonable assumption given that there is an immense push in the industry for standardization efforts for SDI templates [14, 17, 32, 46, 61, 64].

4.3.4 Knob Turning.

The goal of knob turning is to try out various configurations of objects in emulations in order to test the policy's impact on observed variables (e.g., latency, throughput). Since we assume no domain knowledge about what effect the various values of the configurable variables will have on the observed variables, so we use a process of coordinate descent, where a coordinate is a configurable variable, and pick N number of uniformly distributed values for this variable. The uniform distribution allows us to approximately cover the entire range of the variable and is the best approach in the face of no prior knowledge.

These N values become the starter set to begin creating the emulations and trying out the policy within them. As described later, we collect the results of emulation runs and generate a dataset that can be used for creating machine learning models which depict the effect of the starter set on the observed variables. We split the dataset collected and use part of it to train the models, and the other part is used as ground truth to test the accuracy of the models. If the accuracy is lower than a threshold for the starter set, we generate more values through knob turning and collect more data through emulations.

For configuration variables whose range is numeric or ranked i.e., the range of values have some pattern (e.g., EPC. numSGW is a numeric variable, while Server.type can be a ranked variable with the ranking done in terms of numCPUs on each machine type), we can follow the following greedy procedure to more smartly generate the knob turns if the model accuracy for the starter set turns out to be low.

The starter set values for a given variable divide the variable's range into multiple intervals. E.g., if the range of a numeric variable is 1 to 100, and the starter set contains 10, 43, 79, then 3 intervals are created. Set a minimum interval size after which exploration through emulations should stop. For each interval, if the interval size is above the minimum size threshold, then take the dataset collected so far through emulations, and pick M samples from the dataset such that they all belong to the given interval. Use the model created from the dataset so far and input the M samples as feature values and predict the observed variable i.e., the dependent variable in the model. Similarly, calculate the prediction accuracies for the other intervals. Ignore the intervals that have acceptable accuracy. From the intervals with accuracy lower than threshold, pick the widest interval with the highest inaccuracy value and perform the recursive process of picking starter values from this interval and performing emulations to get more data for them, then rechecking the accuracies for all past and newly generated intervals, and exploring the widest interval with the highest accuracy.

This procedure allows us to find the widest intervals that are the most inaccurate according to the calculated machine learning models, thus allowing us to greedily explore in those intervals that will give us the most gain in terms of reducing uncertainty in the model. A stopping condition can be set using parameters for the smallest interval size threshold after which exploration should stop, and/or an accuracy threshold after which the exploration should stop, and/or max number of emulations to run threshold which can be useful since a low accuracy score for the models may not always mean more emulations will help since it might be an issue related to missing variables, or incorrect variable creation by the domain experts, such issues cannot be automatically found since Depo cannot learn about variables that are non-existent for it. Note that when the knob turning process generates a configuration of various knob values, and Depo attempts to generate an emulation environment using those knob settings using the orchestrator, the orchestrator checks the validity of the configuration to see if it can be instantiated.

5 IMPLEMENTATION AND EVALUATION

Environment setup and workload. We extend the SDI orchestrator and common service and object templates from prior work published in the community [12, 61]. We modify the templates with additional mechanisms and variables to allow creation of richer policies. We use Drools [53] for policy specification. For serving as the emulation environment, we created a prototype SDI in the Phantomnet mobility testbed [7, 41] as shown in Figure 7. Our SDI consisted of a combination of server types available in Phantomnet (pc3000, d710, d430, d820) over which virtualized infrastructure and services can be instantiated by the SDI orchestrator as needed for emulations. For workload generation in emulations, depending on extracted service types in the Depo process, we created preset workload generators. These include generators for orchestration and service management request generations e.g., service instantiations, deletions, scaling requests, and also include UE traffic generation based on configurable parameters where the parameters are configured and varied as part of Depo's knob turning.

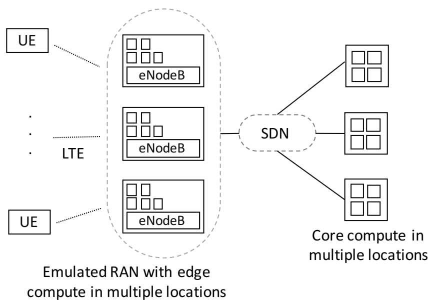
UE
eNodeB
LTE
UE
eNodeB
eNodeB
SDN
Core compute in multiple locations
Emulated RAN with edge compute in multiple locations

Figure 7: Topology in testbed. Multiple locations represent residential, and business and touristy areas.

Policy impact. Listings 1 to 6 show our usecase policies. Depo outputs and learns the trace of each policy as it invokes mechanisms on SDI objects, affecting their configuration variables, which in term impact the observed variables.

E.g., Listing 11 shows the SDI-level edge-server update policy's invoked mechanisms. The policy action had taken the strategy: Divide the servers to be updated in two batches, update one batch at a time by migrating VMs to available servers and stopping the remaining VMs, install updates then restart VM and VNF. Table 2 shows Depo learns the find-grained impact from the various mechanisms invoked by the policy, on objects and their variables. The table captures only those object variables that the policy impacted in a statistically significant way, as seen from calculating p-values and setting their alpha to 0.05 (for 95% confidence). Recording this in the KG allows Depo to know which mechanisms have impact on which variables, which helps Depo in the emulation environment generation phase by reducing the policy's potentially impacted neighbors, meaning that knob turning has to be done on less object variables in newer iterations of emulations. This is also useful for the policy writer since it traces their policy across various SDI objects.

While Table 2 shows a detailed trace useful for the continuous impact checking process, Depo outputs a summary of object variables impacted to the policy writer. E.g., for the SGW scaling policy, Listing 12 is output. Listings 9 and 10 show some more output examples (for SMORE edge cloud offloading related policy for turning on caching, and enabling edge cloud offloading functionality dynamically on EPC).

Table 2: Affected variables learned for update policy.

Mechanisms	Variables
PolicyAction:update	Server.[status, version, numVM, memUsage, cpuUsage, percentFailedMigrations], VM.[status, memUsage, cpuUsage], VNF.[status, memUsage, cpuUsage], Service.[smoreLatency]
Server.migrateVM	Server.[numVM, memUsage, cpuUsage, percentFailedMigrations], VM.[status, memUsage, cpuUsage], VNF.[status, memUsage, cpuUsage], Service.[smoreLatency]
Server.sendVM	Server.[numVM, memUsage, cpuUsage], VM.[status, memUsage, cpuUsage], VNF.[status, memUsage, cpuUsage], Service.[smoreLatency]
Server.receiveVM	Server.[numVM, memUsage, cpuUsage, percentFailedMigrations], VM.[status, memUsage, cpuUsage], VNF.[status, memUsage, cpuUsage], Service.[smoreLatency]
VM.stop	Server.[memUsage, cpuUsage], VM.[status, memUsage, cpuUsage], VNF.[status, memUsage, cpuUsage], Service.[smoreLatency]
Server.installUpdate	Server.[status, version, memUsage, cpuUsage]
Server.reboot	Server.[status, memUsage, cpuUsage]
VM.start	Server.[memUsage, cpuUsage], VM.[status, memUsage, cpuUsage]
VNF.start	Server.[memUsage, cpuUsage], VM.[memUsage, cpuUsage], VNF.[status, memUsage, cpuUsage], Service.[smoreLatency]

Listing 9: Affected variables for various object types in SMORE caching policy use-case.

```
PolicyAction:setCaching
        SMORE:
                smore_cachingStatus
                smore_latency
        SMORE_Loadbalancer:
                smore_loadbalancer_cachingStatus
                smore_loadbalancer_availableCacheSize
                smore_loadbalancer_usedCacheSize
                smore_loadbalancer_resourceFrequencyOfAccess
        SMORE_Webserver:
                smore_webserver_status
                smore_webserver_networkConfig
                smore_webserver_loadbalancerConnectionStatus
                smore_webserver_cpuUsage
                smore_webserver_memUsage
                smore_webserver_resourceFrequencyOfAccess
```

Listing 10: Affected variables for various object types in EPC offloading to SMORE policy use-case.

```
PolicyAction:augmentSMORE
        Server:
                server_numVM
                server_cpuUsage
                server_memUsage
                server_allocated_mem
                server_num_allocated_cpu
        VM:
                vm_status
                vm_networkConfig
                vm_cpuUsage
                vm_memUsage
        SMORE:
                smore_latency
        SMORE_Loadbalancer:
                smore_loadbalancer_webserverNumConnections
                smore_loadbalancer_webserverPercentOfConnectionsWorking
                smore_loadbalancer_resourceFrequencyOfAccess
        SMORE_Switch:
                smore_switch_status
                smore_switch_routeConfig
        SMORE_Webserver:
                smore_webserver_status
                smore_webserver_networkConfig
                smore_webserver_loadbalancerConnectionStatus
                smore_webserver_cpuUsage
                smore_webserver_memUsage
                smore_webserver_resourceFrequencyOfAccess
```

Listing 11: Invoked mechanisms hierarchy for Server update policy.

```
PolicyAction:update
        Server.migrateVM
               Server.sendVM
               Server.receiveVM
        VM.stop
        Server.installUpdate
        Server.reboot
        VM.start
        VNF.start
```

Listing 12: Affected variables for scaleUpSGW policy.

```
PolicyAction:scaleUpSGW
    Server:
          server_numVM
          server_cpuUsage
          server_memUsage
          server_allocated_mem
          server_num_allocated_cpu
    VM:
          vm_status
          vm_networkConfig
          vm_cpuUsage
          vm_memUsage
    EPC:
          epc_numSGW
          epc_latency
    SGW:
          sgw_status
          sgw_networkConfig
          sgw_mmeNumConnections
          sgw_mmePercentOfConnectionsWorking
          sgw_cpuUsage
          sgw_memUsage
    MME:
          mme_numSGW
          mme_sgwNumConnections
          mme_sgwPercentOfConnectionsWorking
```

Continuous learning from multiple emulations using knob turning. In the data analysis for our policies, we tested ML model creation based on polynomial regression, SVM based regression, and boosted decision tree (DT) based regression from scikit-learn and XGBoost libraries [57, 67]. All three of these are for supervised learning as required by the Depo process. We found DT models to perform better than or equivalent to other types of models, in terms of model accuracy (calculated using automated k-fold technique [66] with 80%-20% dataset split into training and test datasets). And so used that as the default model in our prototype.

In order to test the usefulness of performing our knob turning approach when creating emulations, we tested out our three SMORE service related policy examples in Depo using all possible emulation knobs and recorded the generated datasets, we call this set of datasets: the ground truth. This ground truth set is only collected for the purposes of Depo evaluation so that we can evaluate the models generated using the knob turning approach of Depo.

Next, we started a normal policy impact checking test for the three policies using the Depo process. During the process, the first emulation using results in an overall ML model that captures the dependent or observed variable (SMORE response time in our case). Then each new emulation resulting from the knob turning approach causes, on average, improvement of the previously built model. To evaluate that newer emulations actually result in model improvement, we compare the generated model from each emulation iteration, with the ground truth mentioned above. We compare the model accuracy by calculating the error between the model's prediction for SMORE response time, and the ground truth. Figure 8 shows that on average, our knob turning approach causes newer emulations to reduce this error thus showing it improves the models using its greedy approach.

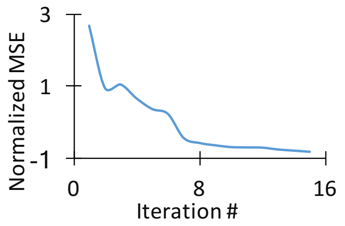

Figure 8: Normalized mean squared error (NMSE) calculated over error between ground truth, and predictions of model created by Depo as it is improved through increasing number of emulation iterations.

Variable reduction and cause of impact. Knowledge-based model learnt from emulations in Depo allows operators to find 'what' was the cause of impact on observed variables of interest. E.g., our Server CPU oversubscription policy invokes mechanisms on different SDI object instances, and causes their configuration variables to change, which ultimately results in impact on observed variables, e.g., SMORE response time.

We use our generated ML models to quantify the impact that each of the invoked mechanisms has on various configuration variable, which in turn affect observed variables. E.g., Table 3 shows relative impact caused on SMORE response

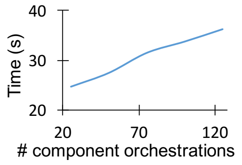

Figure 9: **Orchestration time.**

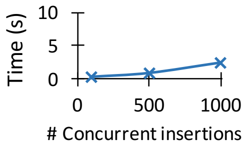

Figure 10: **KG insertion time.**

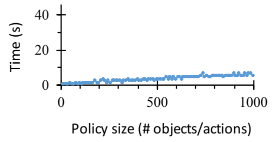

Figure 11: **Policy parsing time.**

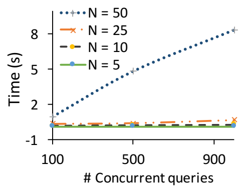

Figure 12: **Parameter generation time.**

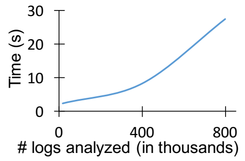

Figure 13: **Data analysis time.**

time by the various configuration variables modified by the policy, the various emulation environment parameters (such as num of neighboring VMs), and the workload variables (e.g., SMORE request rate). Thus, in figuring out the policy impact on a given observed variable, DEPO considerably reduced the potentially impacted object types and variables down from the ones shown in Table 1. The statistical significance testing (with significance alpha value set to 0.05 or 5%) is embedded in DEPO process and helps ensure correctness.

Performance results. Here, we present performance results for each stage of the DEPO process to show the scalability of our approach and use of the KG data structure.

DEPO's policy parsing time is impacted by the policy size, which is dependent on the number of object types and actions in the policy specification. Figure 11 shows this effect for policies specified in Drools [53]. The time stays below 5s for up to a 1000 object types and actions. Since the number of objects and actions in our example policies were small, we show scalability by generating policies using arbitrary object types and actions and running the parser on them.

Figure 12 shows the time for emulation parameter generation using our knob turning approach. As described in the DEPO process earlier, this stage is dominated by time spent on queries into the KG for searching neighbor object types that are potentially under impact radius of the object types parsed from the policy. The time is thus affected by the number of neighboring objects N, that exist in the KG for a given object type parsed from the policy. So, in order to show scalability for this phase, we add increasing number of semantically correct but arbitrary relationships among KG nodes in a controlled fashion. Figure 12 shows the time for parameter generation stage for different values of N. Since multiple object types can be parsed from a given policy, parameter generation stage will have to perform KG queries for extracting neighbors of each of the parsed object types. Thus, the same figure here also shows effect of such concurrent queries on the KG. Our correctness checking assertions showed that the queries are able to retrieve all objects types that had been created as neighbors in the KG.

Table 3: Effect of Server CPU oversubscription policy on SMORE response time.

Variables	Importance
VM.cpuUsage	0.254
VM.numNeighborVMs	0.002
Server.cpuUsage (aggregate)	0.298
Server.oversubscription	0.289
SMORE.requestRate	0.157

Figure 9 shows the emulation environment generation time, which is essentially the SDI orchestration time for the various object types that need to be instantiated for a given emulation. The figure shows time stays below 40s for up to 125 SDI object instance creations, where instances are for various types of object templates available to us and are instantiated and configured in parallel where possible (e.g., a host VM has to be instantiated *before* the VNF it will host).

Figure 13 shows the data analysis stage average time for our policies, which is dominated by the concurrent KS tests and ML model generations. The figure shows the time stays low as number of logs used for the analysis increases by increasing duration and number of emulations. The number of logs depends on the number of object variables related to the policy and the logging frequency. We set this frequency for observed variables (e.g., latency) to be 1s in the SDI monitoring module, and the configuration variable changes that happen through software were logged whenever the software updated them.

Time for final stage of DEPO process is dominated by annotating the KG based on the knowledge learnt in data analysis. Figure 10 shows fact insertions (shown as equivalent to existing fact updates), the time stays below 3s for up to 1000 concurrent insertions.

6 CONCLUSION

We presented DEPO for policy impact checking in an SDI environment. It performs continuous learning of impact using emulations under varying conditions, and takes a statistical and machine learning based analysis approach on the data obtained from emulations. DEPO's data analysis enables it to discover and quantify the impact of policies on SDI objects in the same or different layers.

7 ACKNOWLEDGMENTS

We would like to thank our anonymous reviewers and our shepherd Hongqiang Liu for their valuable comments. This work is supported in part by the National Science Foundation under grant numbers 1302688 and 1647264.

REFERENCES

[1] 2016. ECOMP (Enhanced Control, Orchestration, Management and Policy) Architecture White Paper. ATT Inc. http://about.att.com/content/dam/snrdocs/ecomp.pdf

[2] 5GPPP. 2015. 5G Vision - The 5G Infrastructure Public Private Partnership: the next generation of communication networks and services. https://5g-ppp.eu/wp-content/uploads/2015/02/5G-Vision-Brochure-v1.pdf.

[3] J.G. Andrews, S. Buzzi, Wan Choi, S.V. Hanly, A. Lozano, A.C.K. Soong, and J.C. Zhang. 2014. What Will 5G Be? *Selected Areas in Communications, IEEE Journal on* (2014).

[4] Bilal Anwer, Theophilus Benson, Nick Feamster, and Dave Levin. 2015. Programming slick network functions. In *Proceedings of the 1st acm sigcomm symposium on software defined networking research*. ACM, 14.

[5] AT&T. 2013. AT&T Domain 2.0 Vision White Paper. https://www.att.com/Common/about_us/pdf/AT&TDomain2.0VisionWhitePaper.pdf.

[6] David Baehrens, Timon Schroeter, Stefan Harmeling, Motoaki Kawanabe, Katja Hansen, and Klaus-Robert MÃžller. 2010. How to explain individual classification decisions. *Journal of Machine Learning Research* 11, Jun (2010), 1803–1831.

[7] Arijit Banerjee, Junguk Cho, Eric Eide, Jonathon Duerig, Binh Nguyen, Robert Ricci, Jacobus Van der Merwe, Kirk Webb, and Gary Wong. 2015. Phantomnet: Research infrastructure for mobile networking, cloud computing and software-defined networking. *GetMobile: Mobile Computing and Communications* 19, 2 (2015), 28–33.

[8] Arijit Banerjee, Rajesh Mahindra, Karthik Sundaresan, Sneha Kasera, Kobus Van der Merwe, and Sampath Rangarajan. 2015. Scaling the LTE control-plane for future mobile access. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*. ACM, 19.

[9] Cory Bennett and Ariel Tseitlin. 2012. Chaos monkey released into the wild. *Netflix Tech Blog* 30 (2012).

[10] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.

[11] Xu Chen, Ming Zhang, Zhuoqing Morley Mao, and Paramvir Bahl. 2008. Automating Network Application Dependency Discovery: Experiences, Limitations, and New Solutions.. In *OSDI*, Vol. 8. 117–130.

[12] Junguk Cho, Binh Nguyen, Arijit Banerjee, Robert Ricci, Jacobus Van der Merwe, and Kirk Webb. 2014. SMORE: software-defined networking mobile offloading architecture. In *Proceedings of the 4th workshop on All things cellular: operations, applications, & challenges*. ACM, 21–26.

[13] Cisco. 2015. Vector Packet Processing (VPP).

[14] CORD. 2019. Central Office Rearchitected as a Datacenter (CORD). http://opencord.org/wp-content/uploads/2016/03/CORD-Whitepaper.pdf. Accessed: 10-01-2018.

[15] Bassam Eljaam. 2005. Customer satisfaction with cellular network performance: Issues and analysis. (2005).

[16] ETSI. 2019. Network Functions Virtualisation (NFV). http://www.etsi.org/technologies-clusters/technologies/nfv. Accessed: 10-01-2018.

[17] ETSI. 2019. Open Source MANO (OSM). https://osm.etsi.org/.

[18] Seyed K Fayaz, Tianlong Yu, Yoshiaki Tobioka, Sagar Chaki, and Vyas Sekar. 2016. BUZZ: testing context-dependent policies in stateful networks. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 275–289.

[19] Nick Feamster and Hari Balakrishnan. 2005. Detecting BGP configuration faults with static analysis. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*. USENIX Association, 43–56.

[20] Ronald A Fisher. 1925. Statistical methods for research workers Oliver and Boyd. *Edinburgh, Scotland* 6 (1925).

[21] P. A. Frangoudis, L. Yala, A. Ksentini, and T. Taleb. 2016. An architecture for on-demand service deployment over a telco CDN. In *2016 IEEE International Conference on Communications (ICC)*. 1–6. https://doi.org/10.1109/ICC.2016.7510921

[22] Jose Manuel Navarro Gonzalez, Javier Andion Jimenez, Juan Carlos Duenas Lopez, et al. 2017. Root Cause Analysis of Network Failures Using Machine Learning and Summarization Techniques. *IEEE Communications Magazine* 55, 9 (2017), 126–131.

[23] Google. 2019. Google Inside Search: The Knowledge Graph. http://bit.ly/2cKZ3cO. Accessed: 10-01-2018.

[24] Jessica Gurevitch, Julia Koricheva, Shinichi Nakagawa, and Gavin Stewart. 2018. Meta-analysis and the science of research synthesis. *Nature* 555, 7695 (2018), 175.

[25] J. Gil Herrera and J. F. Botero. 2016. Resource Allocation in NFV: A Comprehensive Survey. *IEEE Transactions on Network and Service Management* 13, 3 (Sept 2016), 518–532. https://doi.org/10.1109/TNSM.2016.2598420

[26] DPDK Intel. 2015. Data plane development kit.

[27] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. 2013. Real time network policy checking using header space analysis. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. 99–111.

[28] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header space analysis: Static checking for networks. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. 113–126.

[29] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P Brighten Godfrey. 2013. Veriflow: Verifying network-wide invariants in real time. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. 15–27.

[30] Been Kim, Rajiv Khanna, and Oluwasanmi O Koyejo. 2016. Examples are not enough, learn to criticize! criticism for interpretability. In *Advances in Neural Information Processing Systems*. 2280–2288.

[31] Himabindu Lakkaraju, Stephen H Bach, and Jure Leskovec. 2016. Interpretable decision sets: A joint framework for description and prediction. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. ACM, 1675–1684.

[32] Linux. 2019. Open Network Automation Platform (ONAP). https://www.onap.org/. Accessed: 10-01-2018.

[33] Zachary C Lipton. 2016. The mythos of model interpretability. *arXiv preprint arXiv:1606.03490* (2016).

[34] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. 2017. Crystalnet: Faithfully emulating large production networks. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 599–613.

[35] Nuno P Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. 2015. Checking Beliefs in Dynamic Networks.. In *NSDI*. 499–512.

[36] Ajay Mahimkar, Zihui Ge, Jia Wang, Jennifer Yates, Yin Zhang, Joanne Emmons, Brian Huntley, and Mark Stockert. 2011. Rapid detection of maintenance induced changes in service performance. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*. ACM, 13.

[37] Ajay Mahimkar, Zihui Ge, Jennifer Yates, Chris Hristov, Vincent Cordaro, Shane Smith, Jing Xu, and Mark Stockert. 2013. Robust assessment of changes in cellular networks. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*. ACM, 175–186.

[38] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P Godfrey, and Samuel Talmadge King. 2011. Debugging the data plane with anteater. In *ACM SIGCOMM Computer Communication Review*, Vol. 41. ACM, 290–301.

[39] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. 2014. ClickOS and the art of network function virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 459–473.

[40] Frank J Massey Jr. 1951. The Kolmogorov-Smirnov test for goodness of fit. *Journal of the American statistical Association* 46, 253 (1951), 68–78.

[41] Mobile Networking Testbed. 2019. PhantomNet. https://phantomnet.org/.

[42] Timothy Nelson, Christopher Barratt, Daniel J Dougherty, Kathi Fisler, and Shriram Krishnamurthi. 2010. The Margrave Tool for Firewall Analysis.. In *LISA*. 1–18.

[43] NGMN Alliance. 2014. 5G White Paper-Executive Version. *White Paper, December* (2014).

[44] Nokia. 2015. Reinventing telcos for the cloud. http://networks.nokia.com/sites/default/files/document/reinventing_telcos_for_the_cloud_white_paper.pdf.

[45] Magnus Olsson, Stefan Rommer, Catherine Mulligan, Shabnam Sultana, and Lars Frid. 2009. *SAE and the Evolved Packet Core: Driving the mobile broadband revolution*. Academic Press.

[46] Open Network Automation Platform. 2017. ONAP Architecture Overview. https://www.onap.org/wp-content/uploads/sites/20/2018/06/ONAP_CaseSolution_Architecture_0618FNL.pdf.

[47] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. 2016. NetBricks: Taking the V out of NFV.. In *OSDI*. 203–216.

[48] M Patel, B Naughton, C Chan, N Sprecher, S Abeta, A Neal, et al. 2014. Mobile-Edge Computing Introductory Technical White Paper. *White Paper, Mobile-edge Computing (MEC) industry initiative* (2014).

[49] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. 2015. The Design and Implementation of Open vSwitch.. In *NSDI*. 117–130.

[50] Ren Quinn, Josh Kunz, Aisha Syed, Joe Breen, Sneha Kasera, Rob Ricci, and Jacobus Van der Merwe. 2016. KnowNet: Towards a knowledge plane for enterprise network management. In *Network Operations and Management Symposium (NOMS), 2016 IEEE/IFIP*. IEEE, 249–256.

[51] Frak Rayal and Joe Madden. 2015. Cloud RAN is a disruptive technology. Here's why. http://www.fiercewireless.com/tech/story/cloud-ran-disruptive-technology-heres-why/2015-01-20.

[52] Saqib Raza, Yuanbo Zhu, and Chen-Nee Chuah. 2011. Graceful network state migrations. *IEEE/ACM Transactions on Networking (TON)* 19, 4 (2011), 1097–1110.

[53] RedHat. 2019. Drools: Rule Language. https://red.ht/2Xo99VQ. Accessed: 10-01-2018.

[54] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. Why should i trust you?: Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. ACM, 1135–1144.

[55] Marko Robnik-Šikonja and Igor Kononenko. 2008. Explaining classifications for individual instances. *IEEE Transactions on Knowledge and Data Engineering* 20, 5 (2008), 589–600.

[56] Jonathan Rodriguez (Ed.). 2015. *Fundamentals of 5G Mobile Networks*. John Wiley & Sons.

[57] SciKit-Learn. 2019. Machine Learning in Python. http://scikit-learn.org/. Accessed: 10-01-2018.

[58] Peng Sun, Ratul Mahajan, Jennifer Rexford, Lihua Yuan, Ming Zhang, and Ahsan Arefin. 2015. A network-state management service. *ACM SIGCOMM Computer Communication Review* 44, 4 (2015), 563–574.

[59] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky HY Wong, and Hongyi Zeng. 2016. Robotron: Top-down Network Management at Facebook Scale. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*. ACM, 426–439.

[60] Aisha Syed. 2019. Proteus in PhantomNet. https://wiki.emulab.net/wiki/phantomnet/proteus. Accessed: 10-01-2018.

[61] Aisha Syed and Jacobus Van der Merwe. 2016. Proteus: a network service control platform for service evolution in a mobile software defined infrastructure. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking*. ACM, 257–270.

[62] Mukarram Tariq, Amgad Zeitoun, Vytautas Valancius, Nick Feamster, and Mostafa Ammar. 2008. Answering what-if deployment and configuration questions with wise. In *ACM SIGCOMM Computer Communication Review*, Vol. 38. ACM, 99–110.

[63] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. 2013. IOFlow: a software-defined storage architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 182–196.

[64] TOSCA. 2013. Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer Version 1.0.

[65] Verizon. 2019. Verizon Service Bundles. https://goo.gl/6iVrfN.

[66] Wikipedia. 2019. k-fold Cross Validation. https://en.wikipedia.org/wiki/Cross-validation_(statistics)#k-fold_cross-validation. Accessed: 10-01-2018.

[67] XGBoost. 2019. Optimized Distributed Gradient Boosting Library. https://xgboost.readthedocs.io/en/latest/. Accessed: 10-01-2018.

[68] Geoffrey G Xie, Jibin Zhan, David A Maltz, Hui Zhang, Albert Greenberg, Gisli Hjalmtysson, and Jennifer Rexford. 2005. On static reachability analysis of IP networks. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, Vol. 3. IEEE, 2170–2183.

[69] Danyang Zhuo, Qiao Zhang, Xin Yang, and Vincent Liu. 2016. Canaries in the Network.. In *HotNets*. 36–42.