Practical Automation for Management Planes of Service Provider Infrastructure

Bingzhe Liu
UIUC

Kuan-Yen Chou
UIUC

Pramod Jamkhedkar
AT&T

Bilal Anwer
AT&T

Rakesh K Sinha
AT&T

Kostas Oikonomou*
AT&T

Matthew Caesar
UIUC

P. Brighten Godfrey
UIUC and VMware

ABSTRACT

Managing service provider infrastructures (SPI) is ever more challenging with increasing scale and complexity. Network and container orchestration systems alleviate some manual tasks, but they are generally narrow solutions, with controllers for specific subsystems that do not coordinate on high-level goals, and fall far short of automating the full range of tasks that engineers face day to day.

We seek to highlight the need for "practical automation" to manage SPIs. Via realistic examples, we argue that practical automation should provide cross-controller coordination, and should work within the reality that many tasks will involve humans. We describe a proof-of-concept system that leverages AI planning to synthesize management steps to move the system towards a goal state. A preliminary implementation shows that our approach can accurately generate plans for complex management tasks, while scalability and modeling diverse controllers remain as future challenges.

CCS CONCEPTS

• **Networks** → **Network management**; • **Computer systems organization** → Cloud computing; *Maintainability and maintenance*; *Reliability*; *Availability*; • **Computing methodologies** → **Planning and scheduling**.

KEYWORDS

practical automation, service infrastructure management, planning and synthesis, intent-based networking

ACM Reference Format:
Bingzhe Liu, Kuan-Yen Chou, Pramod Jamkhedkar, Bilal Anwer, Rakesh K Sinha, Kostas Oikonomou, Matthew Caesar, and P. Brighten Godfrey. 2021. Practical Automation for Management Planes of Service Provider Infrastructure. In *Workshop on Flexible Networks Artificial Intelligence Supported Network Flexibility and Agility (FlexNets'21), August 27, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3472735.3473391

*Work was done while at AT&T.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
FlexNets'21, August 27, 2021, Virtual Event, USA
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8634-0/21/08…$15.00
https://doi.org/10.1145/3472735.3473391

1 INTRODUCTION

Managing service provider infrastructures (SPIs) is challenging. Management tasks, including provisioning, upgrade, troubleshooting, and mitigation, are performed frequently to deal with various events like application deployments, performance degradation and failures. Throughout, engineers need to satisfy high-level intents like global resource utilization, connectivity among components, and service capacity. Furthermore, in order to serve emerging needs like 5G and beyond, service providers are deploying small but numerous "edge" data centers to serve a diverse range of network functions. With such increasing scale and complexity, SPIs are seeking to move away from manual management in favor of orchestration systems like Kubernetes [7] in an attempt to automatically manage their network services.

The holy grail of network management is a "self-driving" infrastructure that operates, optimizes, and fixes itself. However, we are far from that goal in (at least) two respects.

First, current orchestration systems and controllers do not perform global automation. Generally, automated controllers are not aware of combinations of system-wide high-level intents. Instead, each controller has a narrowly-defined task related to a specific subsystem and is not directly aware of other controllers. For example, a load balancer cares about request latency and only controls the distribution of requests, while a scheduler cares about resource utilization across nodes and only controls the placement of the containers. But neither of those encompasses the high-level goal of service capacity. Though they are controlling different portions of the system properties and elements, their local constraints may be intertwined or even conflicting with each other in ways that affect the high-level intents. Hidden shared dependencies among multiple controllers could even lead to failures or potential non-convergence [10].

Second, we argue that many specific management tasks will require humans for the foreseeable future. Automating systems takes time and is typically incomplete; even in a well-designed infrastructure, new sub-systems are incorporated regularly which may not be fully integrated; many tasks require human cross-checking that all is well after a change; and when things go wrong, humans have to troubleshoot. Hence, we expect some human actions will need to coexist with (and interact with) automated controllers.

In this work, we call attention to the need for "practical automation" for the management plane of SPI. By practical, we mean that the overall automation approach takes into account the two characteristic limitations above: (a) the SPI management plane will be comprised of multiple narrowly-tasked controllers that need to be coordinated to work towards high-level management goals; and

(b) humans will be effectively playing the role of some of those controllers.

To motivate the need for practical automation, we discuss the typical properties of modern SPI and orchestration systems, and introduce three representative examples that demonstrate the complexity of management tasks. We then describe a proof-of-concept system, **Strategyzer**, that synthesizes plans for management tasks. The generated plan could be either a recommendation to human operators, or could be automatically deployed by pushing instructions to controllers. Such planning is non-trivial because (1) we need to model the complex interactions among controllers and (2) the expected management tasks normally consist of a sequence of steps rather than a simple change of configurations, resulting in a large search space. To deal with these challenges, we utilize technology from the area of AI Planning [3], which aims to automatically synthesize a sequence of actions to achieve the given goals. We model both human actions and controllers using a formal language, where their interactions could be reflected by clearly defined pre- and post-conditions. We implemented two management examples as case studies and show that the approach can generate plans accurately for these tasks. However, while Strategyzer is useful as a proof-of-concept of functionality, it is far from a complete system. We close with a discussion of scalability and other challenges for future work to achieve practical automation for SPI.

2 BACKGROUND

In this section, we first provide an overview of service provider infrastructure in §2.1. Then we introduce controllers in the management planes of SPI in §2.2, where we treat human as one type of controller. We finally describe the limitations of the current systems and summarize with goals of our systems in §2.3.

2.1 Overview of Service Provider Infrastructure

Network service providers offer a wide range of network services including VPN, CDN, SD-WAN, and so forth [1]. The infrastructure supporting these network services consists of several subsystems, including the data plane which contains network equipment (e.g. routers, switches, eNodeBs), network automation controllers which manage various aspects of the network (e.g. handling network failures, managing traffic congestion), and container orchestration systems (such as Kubernetes and Openshift [7, 14]) which manage the network automation controllers and other software components that are deployed as containers. These different subsystems typically reside in and across multiple sites and edge data centers, which are gaining more popularity over the years due to increased usage of online services, growing demand for edge computing, and rising demand for high-bandwidth services [15].

SPIs are required to meet high-level service intents such as availability, connectivity, and service capacity even facing failures and performance degradation. To comply with these intents and handle operational events, the management planes of SPI involve various types of tasks, such as provisioning, upgrade, troubleshooting and failure mitigation. These tasks can be carried out by both automated controllers and human actions which we describe in §2.2. The tasks typically involve sequences of actions. For example, one team first needs to manually fix the RAN equipment on-site, before

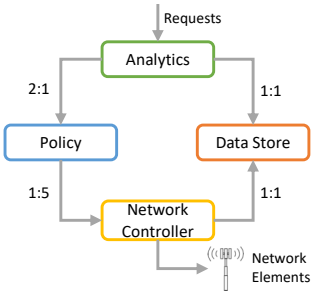

Figure 1: A simplified, representative architecture for network automation controller (deployed as services).

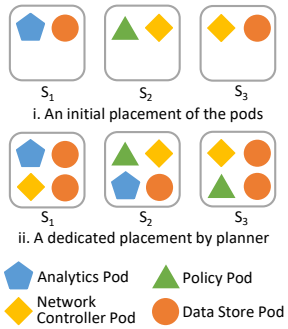

Figure 2: Placement of sub-components.

a remote configuration change can be executed by another team. In many cases, the task requires domain knowledge of networks and RAN equipment, and cross-team collaboration. These actions might theoretically be automated, but in reality, most SPIs are hybrid and require some level of human intervention. Tasks also take into consideration the *cost* of an action; for example, a fast hard reset may be acceptable for one service, but may be unacceptable for another service with stricter requirements on availability.

2.2 Controllers in Service Provider Infrastructure

Controllers act as control loops to watch the state of the infrastructure, dynamically react to system changes and maintain the desired states. Controllers can be either automated (software) or human (manual actions) § 2.2.3. Automated controllers can be further divided into two types: 1) network automation controllers in § 2.2.1 that manage network elements; and 2) controllers in container orchestration systems in § 2.2.2. Though we describe network automation controllers, in this paper, we mainly focus on the controllers in container orchestration systems and human "controllers", and we leave the network automation controllers as future work.

2.2.1 Network Automation Controllers. Network automation controllers control network elements including routers, switches, and RAN equipment (e.g. eNodeBs, sectors, cells). These can include virtual network functions (VNFs) for configuring routers, firewalls, NAT services, or even more sophisticated controllers which manage thousands of network devices to handle traffic congestion and optimize network resource.

Figure 1 shows a simplified, representative architecture for network automation controllers along the lines of the Open Network Automation Platform (ONAP)[1] [12]. These controllers consist of several sub-components as represented by nodes in the graph in Figure 1, and the edges represent the communication between sub-components. Each sub-component is implemented as an independent service deployment on Kubernetes, consisting of one or more pods[2] that run on nodes[3] in the cluster.

[1] While ONAP provides a representative architecture for subsequent discussions in the paper, the principles and results presented in the paper can be applied to generic service provider infrastructure architectures.
[2] A pod is the basic execution unit that runs one or more containers and their associated resources.
[3] A node is typically a virtual or a physical machine that serves pods.

Consider, for example, a load balancing network controller with a similar structure as Figure 1, which manages traffic in RAN for improved throughput. The Data Store (DS) collects and stores information about RAN elements and their KPIs; Analytics then queries the DS for these KPIs, analyzes them, and identifies the RAN elements that need to be configured; the identified RAN elements are then passed on to Policy to determine if it is safe to reconfigure them based on rules like maintenance status, time of day and market; Policy then directs the Network Controller to reconfigure RAN elements via south-bound APIs. Other controllers (e.g. handling network congestion, resource saving) consist of similar workflows. These network controllers change the system state in complex ways, and we leave them for future work.

2.2.2 Container Orchestration Controllers. Container orchestration platforms enable automated management of the network automation controllers that are deployed as services. Some representative controllers are described below, and we define some notations of controller configurations that we will use in the rest of the paper, where the actual configurations can be more complex than simple notations.

Load balancer (LB) distributes input traffic to a pool of nodes using various mechanisms, e.g. round-robin and weighted round-robin. In weighted round-robin, each node is dynamically assigned a weight according to a combination of changing performance metrics, e.g., number of active connections.

Deployment controller controls pod deployment, update, runtime and termination in a deployment. A commonly used configuration is to maintain a certain number k of pods on a particular node by configuring `specs.replica = k`.

Scheduler places unscheduled pods onto the best feasible nodes according to user-defined policies. For example, `specs.CPU = 0.7` specifies scheduling pods on a node only when the expected CPU usage is less than 70% after pod placement.

Descheduler [8] evicts pods from nodes according to node resource usage and operator-defined strategies. For example, configurations `specs.CPU = 0.8` and `spec.MostUsed` denote that the descheduler should evict the pod with highest CPU usage when CPU utilization is over 80% on that node.

2.2.3 Human "Controllers". More often than not, certain tasks for infrastructure management are left to humans for many reasons. These include legacy components which don't have automation support such as older network equipment which requires CLI-based configurations. Failures at physical level require manual intervention such as fixing a broken cable, replacing a dead hard drive, or an onsite fix of network card on the interface between LTE RAN and Evolved Packet Core. Sometimes humans may be involved due to cost concerns in automating a wide range of equipment management from multiple vendors. These tasks form a critical part of troubleshooting processes and management of SPI. In our system, we model these human actions similar to how automated controllers are modeled. This broadens the scope of our system to more realistic troubleshooting strategies.

2.3 Summary of Current Limitations

Management planes that currently consist of controllers described in § 2.2 have the following limitations.

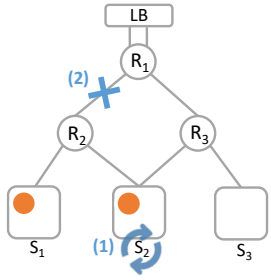

Figure 3: Case 1, (1) human take down S_2 to update, and (2) link $R_1 - R_2$ failed. Both the pods are unavailable.

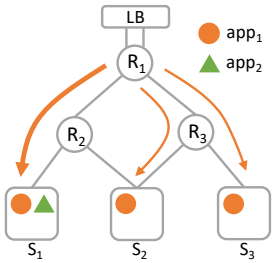

Figure 4: Case 3, pod placement and path choices.

Limited context. Automated controllers' context is localized as they do not consider system-wide high-level goals, e.g. service-level intents. Service requirements include performance specifications and service level agreements (SLAs) which are broader than the scope and capabilities of individual controllers.

Limited action coverage. Due to limited context of controllers, requirements of a service may not be fully expressible or executable in terms of the controllers. For example, ensuring service availability may require considering failures outside the domain of controllers, such as failures of communication links between the nodes. Fixing these failures may require human intervention, such as fixing physical wiring. Hence any realistic handling of a service failure, more often than not, requires a collaborative effort among the controllers including human "controllers".

Local optimization. Since controllers operate independently and optimize local goals, we may end up in a situation where one controller's actions negate the effect of another. These limitations are exacerbated when multiple services are sharing a platform.

In summary, a practical automation management system should address these limitations with following features: 1) ability to consider the interactions between controllers; 2) ability to ingest high-level goals and propose plans to persistently satisfy those goals in case of service failures and service degradation; 3) ability to propose efficient plans according to operational costs.

3 MOTIVATING EXAMPLES

In this section, we describe three examples that illustrate the challenges mentioned in §2.3, and show the need of practical automation for the management plane of SPI.

3.1 Case 1: Integrating Human Actions with Controllers and Environmental Events

SPI often requires software and security updates to pods or nodes. Managing these updates on a live system serving applications is a challenging task. The updates are typically planned to ensure overall service availability, however, an unplanned environmental event such as a link failure occurring during the process can render the service inaccessible.

In Figure 3, three servers are connected to an LB through a network of three routers. One application deployment with 2 pod replicas have been placed on nodes S_1 and S_2. The infrastructure

team aims to update software across all the nodes. To update a node, they need to manually take down the node, install software, and bring it back up, during which all the pods on that node become unavailable. Only one node is updated at a time. Besides these manual update actions, a deployment controller has set `specs.replica = 2` to maintain the 2 pods, and a scheduler has set `specs.CPU = 0.8`. (Other controllers are not involved.) Meanwhile, environmental events could bring down up to 1 link at non-deterministic points of execution. In this update process, high-level intents are (a) at least 1 replica should be up and reachable continuously, (b) all nodes should be updated, and (c) resource usage constraints should be respected.

It's not trivial to plan for a safe update event. Without careful planning, as shown in Figure 3, the following events may happen in sequence that violate the high-level intents: (1) Humans take down S_2 to start the update. (2) Link $R_1 - R_2$ goes down, and the replica on S_1 becomes unreachable. (3) None of the replicas are available and the high-level intent (a) is violated. Note that though the deployment controller aims to maintain 2 pods, it doesn't consider environmental events and hence fails to prepare extra pods before link failure.

A safe plan should consider the potential environmental event and the context of the update process, and schedule one more pod onto a node that is independent (in terms of shared physical network links) from other replicas. A safe plan would: (1) Instruct the deployment controller to increase `specs.replica` by 1. (2) Instruct the scheduler to place the new pod on to node S_3. (3) Humans start to update S_1, S_2 and S_3 accordingly. In this plan, the controllers are instructed to coordinate with each other as well as human actions.

3.2 Case 2: Service-level Intents

Network automation controllers can be represented as a directed acyclic graph and deployed as services as introduced in §2.2.1. In Figure 1, each node n_i represents a sub-component i, and each edge $n_i \rightarrow n_j$ represents the message flows from n_i to n_j. The edge is associated with a message ratio $p\colon q$, which denotes that for each p messages n_i receives from its upstream components, q messages would be generated and sent to its downstream component n_j (for simplicity, we think of one message requiring one unit of work). The component lacking upstream components is the front-end to which user requests arrive.

Service capacity `capacity(S) = k` is a service-level intent saying that service S should be able to process k user requests per second. This requirement needs to be translated into lower-level instructions, such as to maintain a certain number of pods in each deployment.

We use the graph in Figure 1 for this case. To translate the capacity intent, let's first assume that each pod could process 20 requests per second (rps) regardless of its component type, and we expect `capacity(S) = 10`. In order to process 10 rps, Analytics needs to process the 10 original service requests, and hence needs 1 pod. It then generates 5 rps to Policy and 10 rps to Data Store. Policy only receives requests from Analytics, and hence needs to process 5 rps and requires 1 pod. Policy then generates 25 rps to the Network Controller and so on. The result is Network Controller needs 25 rps and requires 2 pods, and Data Store needs 35 rps and requires 3 pods. Figure 2 shows their initial placements.

The planning happens when two different human teams aim to do their own tasks simultaneously. Similar to Case 1, an infrastructure team aims to update all the nodes and take down at most 1 node at one time. Meanwhile, an application team aims to increase service capacity to 15. Here, two controllers are relevant: the deployment controllers and a scheduler. The high-level intents in this planning are to (a) change `capacity(S)` to 15, (b) update all the nodes, and (c) maintain resource usage constraints.

An automated planner would first calculate the number of pods that need to be changed to meet with the new intent. In particular, the number of Data Store pods must be increased by 1. Then, the planner calculates the number of extra pods and places them carefully to ensure service capacity during software updates. Figure 2 shows the expected placement pre-update. Finally, the planner instructs the deployment controllers and scheduler to place the pods as expected, and then instructs the infrastructure team to start updates.

Note that while past work has studied service resource management, the new need here is to plan in the context of other moving parts like manual software updates.

3.3 Case 3: Efficient Controller Ordering

In this case, the network topology is shown in Figure 4. The routers use equal-cost multi-path routing (ECMP) with destination hashing. Another two controllers participate in this example: a descheduler with `specs.CPU = 0.8` and an LB that runs a weighted round-robin mechanism according to service response time. Service response time depends on server-side latency and the accumulated link latency along the path between the server and the LB. There are two applications: app_1 has 3 replicas and app_2 has 1 replica. Figure 4 shows the placement of the replicas and the ECMP path choices for app_1. The high-level intents are (a) maintaining resource usage constraints (80% CPU limit in this case), (b) optimizing request latency, and (c) minimizing cost of the plan in terms of pod migration.

Everything is stable initially. Then, a surge of requests to app_2 causes resource usage on S_1 to exceed the threshold. Two controllers can react to this event to reduce the load on S_1: a descheduler that evicts a pod on S_1, or an LB that re-balances the app_1 requests to the replicas S_2 and S_3 as it sees increased request latency from S_1. Because they involve pod movement, descheduling and rescheduling are more costly than re-balancing requests. As the controllers are not coordinating with each other, a less preferred situation may happen when the descheduler reacts first. If planning is coordinated, a more efficient sequence would be to instruct the descheduler to pause and wait for the LB to re-balance the app_1 requests among the replicas, and to resume after LB finishing its tasks, only if needed.

4 TOWARDS PRACTICAL AUTOMATION

We introduce a proof-of-concept system, Strategyzer, as a first step towards practical automation for the management plane of SPI. Our goal is not to propose a new formal technique, but to introduce an existing planning mechanism to our domain, which brings both opportunity and challenges.

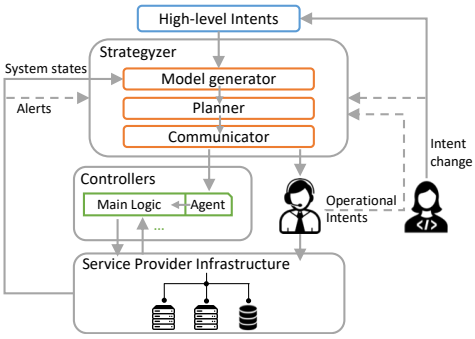

Figure 5: Strategyzer workflow.

4.1 System Design

Figure 5 shows an overview of the Strategyzer workflow. The management plane of SPI consists of automated controllers and human "controllers". The infrastructure monitors and stores various system metrics (e.g. resource usage, node status), and may generate both normal events (e.g. a new pod need to be scheduled) and abnormal events as *alerts* (e.g. CPU usage above a threshold). The controllers continuously monitor the underlying infrastructure and react to metric changes or ongoing normal events. There may be different human teams, e.g. an infrastructure team and multiple application teams, which take care of various parts of a system.

Strategyzer may be triggered when high-level intents change, new operational events are expected (e.g., software update), or the infrastructure generates alerts (dashed lines in Figure 5). There are three main components of Strategyzer: a *model generator* takes current system states and high-level intents as input and generates a formal model; a *planner* uses the model to generate a sequence of management actions; and finally a *communicator* pushes instructions to automated controllers and assists human operators according to the plan. The plan can be a suggestion to human operators, or can be automatically deployed through the communicator. Each automated controller has an agent that continuously listens to the communicator, and when facing an event, pauses its automated reaction to wait for instructions from its agent via the communicator.

In this work, we mainly focus on the modeling and planning techniques assuming the system states have been collected. We leave the implementation of remaining parts (including the agent on controllers) to future work. In addition, we assume operators provide us the knowledge of human actions and automated controllers. The potential insufficiency caused by human intervention and the study of appropriate separation between human and automation are out of scope of our work.

4.2 Modeling Planning Problem

We leverage the area of Artificial Intelligence (AI) planning to automatically synthesize steps to achieve given goals.

We need to model two key parts of the system: (1) controllers, both automated and human; and (2) system elements and their states, e.g. pods and their placements, nodes and their resource usage. The Planning Domain Definition Language (PDDL) [4] is a formal knowledge representation language designed to express planning model in AI planning. We leverage PDDL and model each controller with pre- and post-conditions. For example, for the descheduler, pre-conditions include that CPU usage is above a threshold on a node, and post-conditions include that the pod with highest CPU usage is removed from the node. This representation could help us to focus on reasoning about the interactions between controllers, rather than complex implementation details. The system elements are defined as objects in PDDL and their states are defined by Boolean predicates (e.g. node up/down) and a function that maps objects onto numeric domains (e.g. number of pods on a node). Modeling the system is not always straightforward, and often requires special treatments, e.g. preprocessing to translate high-level intents into appropriate constraints. We give a detailed example in the study of Case 1 in § 4.3.

We are interested in high-level intents including both *safety* and *liveness* properties. The safety properties denotes that something bad will never happen, which requires certain constraints to be satisfied all the time, e.g. the number of replicas should always be more than 1. The liveness properties states that something good will eventually happen, which requires the states to be transformed to end goals, e.g. all the nodes need to be updated. PDDL provides the definition of state trajectory constraints using linear temporal logic such as `always`, and we leverage such constraints to describe the safety properties. The liveness properties are defined as goals in PDDL.

4.3 Case Study Implementation

We briefly describe our implementation of two motivating examples introduced in § 3. All the following examples are implemented in PDDL and evaluated by a numeric planner Metric-FF [5] on an Intel Xeon E5-2697 with 28 cores and 189GB RAM running Linux 5.11.16.

Safety+liveness: software update + environmental events. We implement Case 1 from § 3.1. This case needs to consider non-deterministic environmental events which can apply to any link in the topology and at anytime during the software update. Modelling such non-determinism is non-trivial and most planning tools cannot fit our needs. We translate the potential environmental events into additional constraints in the planning model. Each failure event can isolate a subset of nodes (i.e., they are no longer reachable from users and the rest of the network), which we refer to as a *shared-risk group (SRG)*. When considering up to 1 link failure, for the simple topology in Figure 3, the SRGs are S_1 and S_3. For the high-level intent (a) in § 3.1, the constraint can be defined as: for all the SRG_i, the number of available replicas in the rest of topology (i.e. except for the replicas in the nodes of SRG_i) should be at least 1.

Optimization: generating efficient plan. We implement Case 3 in § 3.3. To find an efficient plan, we assign a cost to each action and add an optimization goal of minimizing total cost through the `metric` keywords in PDDL. In this case, descheduling has higher cost than balancing the load (i.e. assigning cost 1 for load balancer and 20 for scheduler).

Evaluation and discussion on scalability. Figure 6 shows the experimental results for the two cases, where the y-axis shows the planning time (model generation time is negligible). The graph for the experimental result of memory usage is similar as Figure 6. The topology we used is as following: a core router is connecting to edge routers that can be scaled up, and each edge router connects

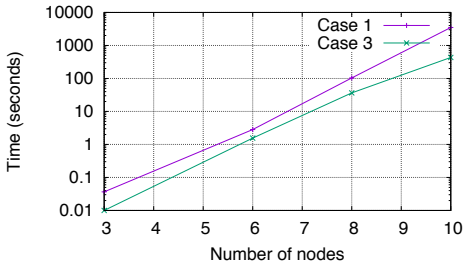

Figure 6: Evaluation results.

to two nodes, where the two nodes form a SRG for 1 link failure. For Case 1, we scale up the number of nodes, and the initial placement is on two different SRGs. For Case 3, we scale up the number of nodes and pods for app_1, and place each pod replica onto a separate node. In both cases, the rest of the setup remains unchanged.

We manually inspect the generated plans and they all accurately result in the goal state. Case 3 has better performance than Case 1, because it has a much shorter series of actions (around 5 steps in all scenarios) than Case 1 (which grows linearly with number of nodes and is 144 steps for 10 nodes).

While the plans are correct, the performance results show this implementation does not scale well. This highlights the challenge of this problem space. To plan an action sequence, the search space is much larger than, say, synthesizing a static configuration. Even with 10 nodes and 3 pods (2 already placed and 1 to be scheduled by a planner), there are around 80 candidate actions at each step (as there are several possible parameterized actions applied to each object in the system, along with "helper" actions that modify internal system state variables). Since the final plan at this scale has 144 steps, a brute force search would inspect very roughly 80^{144} possible plans. Also, we note the planner software package Metric-FF we use was built for a different domain, and so we expect there are domain-specific optimizations that can assist.

5 DISCUSSIONS AND FUTURE WORK

Customized planners/solutions for scalability and non-determinism. Our proof-of-concept demonstrated the idea of planning, but we believe a custom domain-specific planner will be necessary to achieve acceptable performance; among other things, it must support a numeric domain, iterative operations like sum, and representing non-determinism. With respect to the latter, we showed how to model non-determinism for one specific application in § 4.3, by including failures which happen at an unknown time as a continuously-preserved resilience constraint. Other variations, e.g. randomness in controllers, may require a customized solution.

In addition to a new planner, preprocessing methods could further optimize the size of the model, e.g. carefully eliminating the objects absent from a planning event. For planning in the face of failures, use of SRGs along the lines of our prototype may be key: for example, Amazon has only 80 availability zones, while modeling thousands or millions of individual network links is impractical and adds little value.

Formal modeling of controllers and human actions. Currently experts need to manually model controllers in PDDL. Although this is one of the challenges in our system, we believe there is hope. Controllers in container orchestration are becoming standardized, and one controller can have a unified specification across different orchestration systems. We believe modeling these controllers can be a one-time effort. For network automation controllers in SPI, one can develop a domain-specific language to provide a generalized understanding for different types of controllers and service providers. For human actions, one of the future works is to leverage the approaches from the human-computer interaction (HCI) field to work with operators and summarize a representative lists of management actions.

Controllers in hierarchy. As network automation controllers run as services and are managed by orchestration system, the orchestration controllers become the "controllers of controllers". This may bring new challenges that we plan to explore in the future work. The management system needs to plan under intertwined and dependent goals across different subsystems. For example, if the service capacity goal of network automation controller can't be met, the performance goal in RAN may not be met since the controller may not be able to optimize traffic in time; while the service capacity goal is related to the resource usage goal in orchestration system. Additionally, the orchestration controllers need to appropriately balance the resources between multiple network controllers if they share the same platform.

6 RELATED WORK

Intent-based systems. Google's Orion SDN controller [2] uses intents for updating the network design and adding new features. VISCR [11] automatically detects rogue policies, policy conflicts, and automation bugs. Detecting policy conflicts is an interesting future extension of our work. [6] translates intents expressed in natural languages into a formal policy. These systems are complementary to our problem space because they do not synthesize plans for management tasks.

Microservice management. FIRM [13] considers high-level SLO intents when managing resources in microservices. Gandalf [9] aims to safely deploy software rollouts to the infrastructure. These works can automate a specific perspective of the management, but don't consider various high-level intents like service capacity. [10] describes cross-controller interactions, but it focuses on verification rather than planning.

7 CONCLUSION

Our motivating examples demonstrate the needs for a practical automation for service provider management planes. Our proof-of-concept system can accurately generate plans for complex task, while scalability remains a challenge. A scalable planner that can coordinate across a diverse range of individual controllers is an exciting way to avoid mistakes, reduce human effort, and improve overall service reliability in future service provider networks.

Acknowledgements. We thank Dr. Vijay Gopalakrishnan for his valuable feedback. This material is based upon work supported by the National Science Foundation under grant No. CNS-1513906 and the Maryland Procurement Office under Contract No. H98230-18-D-0007.

REFERENCES

[1] AT&T. AT&T Business Service Guide. https://serviceguidenew.att.com/sg_libraryCustom.

[2] A. D. Ferguson, S. Gribble, C. Hong, C. E. Killian, W. Mohsin, H. Mühe, J. Ong, L. Poutievski, A. Singh, L. Vicisano, R. Alimi, S. S. Chen, M. Conley, S. Mandal, K. Nagaraj, K. N. Bollineni, A. Sabaa, S. Zhang, M. Zhu, and A. Vahdat. Orion: Google's software-defined networking control plane. In J. Mickens and R. Teixeira, editors, *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021*, pages 83–98. USENIX Association, 2021.

[3] H. Geffner and B. Bonet. *A Concise Introduction to Models and Methods for Automated Planning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2013.

[4] P. Haslum, N. Lipovetzky, D. Magazzeni, and C. Muise. *An Introduction to the Planning Domain Definition Language*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2019.

[5] J. Hoffmann. The metric-ff planning system: Translating "ignoring delete lists" to numeric state variables. *J. Artif. Intell. Res.*, 20:291–341, 2003.

[6] A. S. Jacobs, R. J. Pfitscher, R. A. Ferreira, and L. Z. Granville. Refining network intents for self-driving networks. In *Proceedings of the Afternoon Workshop on Self-Driving Networks, SelfDN@SIGCOMM 2018, Budapest, Hungary, August 24, 2018*, pages 15–21. ACM, 2018.

[7] Kubernetes. Kubernetes: Production-grade container orchestration. https://kubernetes.io/, May 2021.

[8] Kubernetes-sigs. Descheduler. https://github.com/kubernetes-sigs/descheduler, June 2020.

[9] Z. Li, Q. Cheng, K. Hsieh, Y. Dang, P. Huang, P. Singh, X. Yang, Q. Lin, Y. Wu, S. Levy, and M. Chintalapati. Gandalf: An intelligent, end-to-end analytics service for safe deployment in large-scale cloud infrastructure. In R. Bhagwan and G. Porter, editors, *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 389–402. USENIX Association, 2020.

[10] B. Liu, A. Kheradmand, M. Caesar, and P. B. Godfrey. Towards verified self-driving infrastructure. In B. Zhao, H. Zheng, H. V. Madhyastha, and V. N. Padmanabhan, editors, *HotNets '20: The 19th ACM Workshop on Hot Topics in Networks, Virtual Event, USA, November 4-6, 2020*, pages 96–102. ACM, 2020.

[11] V. Nagendra, A. Bhattacharya, V. Yegneswaran, A. Rahmati, and S. R. Das. An intent-based automation framework for securing dynamic consumer iot infrastructures. In Y. Huang, I. King, T. Liu, and M. van Steen, editors, *WWW '20: The Web Conference 2020, Taipei, Taiwan, April 20-24, 2020*, pages 1625–1636. ACM / IW3C2, 2020.

[12] T. L. F. Projects. Open network automation platform. https://www.onap.org/, May 2021.

[13] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer. FIRM: an intelligent fine-grained resource management framework for slo-oriented microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 805–825. USENIX Association, 2020.

[14] RedHat. Red hat openshift. https://www.openshift.com/, May 2021.

[15] P. Wadhwani and S. Gankar. Edge data center market share 2020-2026. Global Market Insights, September 2020.