

# A Composition Framework for Change Management

Ajay Mahimkar, Carlos E Andrade, Rakesh K Sinha, Giritharan Rana  
AT&T, USA

{mahimkar, cea, rksinha, giri}@att.com

## ABSTRACT

Change management has been a long-standing challenge for network operations. The large scale and diversity of networks, their complex dependencies, and continuous evolution through technology and software updates combined with the risk of service impact create tremendous challenges to effectively manage changes. In this paper, we use data from a large service provider and experiences of their operations teams to highlight the need for quick and easy adaptation of change management capabilities and keep up with the continuous network changes. We propose a new framework **CORNET** (COmposition fRamework for chaNge managEmenT) with key ideas of modularization of changes into building blocks, flexible composition into change workflows, change plan optimization, change impact verification, and automated translation of high-level change management intent into low-level implementations and mathematical models. We demonstrate the effectiveness of CORNET using real-world data collected from 4G and 5G cellular networks and virtualized services such as VPN and SDWAN running in the cloud as well as experiments conducted on a testbed of virtualized network functions. We also share our operational experiences and lessons learned from successfully using CORNET within a large service provider network over the last three years.

## CCS CONCEPTS

• **Networks** → **Network management; Network manageability; Network dynamics; Network performance analysis; Network experimentation.**

## KEYWORDS

Network change management, composition framework, change plan optimization, impact verification

### ACM Reference Format:

Ajay Mahimkar, Carlos E Andrade, Rakesh K Sinha, Giritharan Rana. 2021. A Composition Framework for Change Management. In *ACM SIGCOMM 2021 Conference (SIGCOMM '21), August 23–28, 2021, Virtual Event, USA*. ACM, New York, NY, USA, 19 pages. <https://doi.org/10.1145/3452296.3472901>

**Ethical issues:** This work does not raise any ethical issues.

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 ACM 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](mailto:permissions@acm.org).

*SIGCOMM '21, August 23–28, 2021, Virtual Event, USA*

© 2021 Association for Computing Machinery.

ACM ISBN 978-1-4503-8383-7/21/08...\$15.00

<https://doi.org/10.1145/3452296.3472901>

## 1 INTRODUCTION

*To improve is to change; To be perfect is to change often.*

–Winston Churchill

Networks are continuously changing [30, 58] in order to support the ever increasing surge in applications such as high definition video, virtual reality, emergency services and to provide an enriched service quality of experience. Network changes take the form of software upgrades, hardware changes, configuration changes, new feature activations, capacity improvements, or introduction of new technologies such as 5G, virtualization, containerization, and software defined networking. All changes come with a risk. If the change is not carefully managed, it can result in tremendous negative impact to user experiences [1–3, 5, 6, 9] such as unavailable service, or poor quality and potentially significant loss of business revenues and reputation to service providers.

Change management is extremely challenging and has been a long-standing problem for network operations teams because of the large scale and diversity of the networks, multiple physical, virtual, and containerized network functions (NF), cross-layer (e.g., physical host to virtualized network functions) and service-layer (e.g., a top-of-rack switch serving multiple application servers) complex dependencies, dynamic topologies, and multiple vendors. Change management tasks comprises of four phases: (i) *design* of a change and steps involved in implementing the change, (ii) *planning* the change deployment while ensuring available capacity to carry over the traffic and co-ordination across different network changes, (iii) *execution* of the change on individual network function instances with decision points for roll-back in case of unexpected fall-outs, and (iv) *verifying* the impact of the changes. Each of the phase has to be managed with extreme caution.

**State of the art.** The change management work has evolved over time with varying degrees of automation and robustness across the four phases. NetCraft [37] and Jinjing [59] from Alibaba proposes new frameworks to automate network configuration changes using a unified network model, verification and synthesis. Robotron [56] from Facebook enables operations teams to express network management objectives using a high-level design intent and converts automatically into low-level device configurations that are deployed safely across the network. In [30], Govindan *et al.* automate their change deployments and carefully monitor network operational invariants. DEPO [57] presents an end-to-end framework to define operations policies, test their impacts in emulation environments, and safely deploy them in production networks. Statesman [55] proposes network management services to deploy changes by carefully accounting for conflicts and network capacities in run-time. Corybantic [44] supports modular composition of controller modules and conflict management over higher-level network deployment policies and objectives. ONIX [34] provides a platform with programmatic access to the network so that any

control plane can be implemented on top of it. PRESTO [24] focuses on automating network configuration management and proposes composable config-lets to implement the changes. There are also several research papers that address configuration synthesis [10, 16–18, 23, 49, 51] to map high-level configuration intent to low-level node-centric configuration, configuration verification using formal methods [14, 25, 27, 28, 32, 33, 42, 48, 54], performance impact verification [13, 36, 39–41, 46, 52, 52] using statistical techniques, and network change schedule planning [11, 31, 38, 50, 55, 60] to avoid run-time conflicts and minimize the risk.

Each of the above solutions can be viewed as custom solutions. They work beautifully for the specific network functions or services (e.g., routers, switches or data center networks) but they may require significant work to extend to new functions and services (e.g., base stations, virtualized gateways, or 5G radio access networks). We also observe similar patterns across operations teams managing different parts of the networks (e.g., edge versus core), different types of changes (e.g., software upgrades versus configuration changes) and even different vendors for the same network function. A change is typically implemented using a sequence of steps captured using a MOP (method of procedure). Each team implements and maintains their own automated solutions, but they cannot be easily re-used. We observe hundreds of different network functions in large operational networks and creating custom solution for each of them is a daunting task. Further, the change planning and co-ordination is also extremely labor intensive across individual silos. Finally, the risk of change impact is modeled after individual network functions but not across groups (e.g., will there be an impact if change is implemented on an access router as well as a connected switch in the transport network?). Given the pace at which network evolves, one has to update the change design and planning procedures, roll-back mechanisms and verifications rules not just for the new network functions, but possibly also for existing network functions. As an example, with 5G technology being rolled out, it is important to verify service performance impacts across both 4G and 5G. Thus, there is a strong need for the network operations teams to keep up with the continuous network evolution. The key challenge facing them today is to be able to easily and quickly adapt their change management capabilities.

**Our approach and key ideas.** Our goal is to simplify all aspects of change management, enable network operations teams to quickly accommodate changes to their deployment requirements and evolve on an ongoing basis. We propose a novel framework CORNET (COmposition fRamework for chaNge managEment) to tackle this goal. Our key ideas are (i) **Modularization** of the change method of procedure into reusable library of building blocks. We define and store the change building blocks into a catalog and support their implementation using software packages such as Ansible [12] playbooks, NetConf [45], Chef [20] recipes, or vendor scripts. The catalog enables the operations teams to easily select and adopt across different change activities and network function types. (ii) **Design composition** (or, stitching) of the building blocks into a change workflow (MOP). CORNET automatically deploys change workflows into Camunda [19] for orchestration and transforms the high-level intent of change design into low-level implementation of the building blocks. It also monitors the success/failure of the

workflows as well as the individual building blocks and respective roll-backs. (iii) **Change plan optimization** for dynamically composing and translating high-level change plan intent into low-level mathematical models and optimization solvers in MiniZinc [43]. The optimization objective in CORNET is to minimize the total change completion time and conformance to change deployment intent. (iv) **Verification** composition of the change impact rules across multiple performance indicators and time intervals based on the intent of the change.

One can view the concept of change composition (or, chaining) analogous to service chaining [15, 29, 61]. We believe we are the **first** to propose a dynamic composition framework for change management. A key advantage of our composition capability in CORNET for the operations teams is that they can maximize the re-use of the building blocks across different change activities and quickly adapt to new change management goals (e.g., new deployment intent, or verifying new performance improvements). The dynamic functionality enables them to quickly adapt their composition depending on the intent of the change deployment. We implement CORNET such that the operations teams can focus on the change intent and easily leverage the user interfaces to specify and manage their deployment goals.

**Our contributions.** Using real-world data and several years of operations experiences (Section 2), we highlight the need for easy and effective management of changes across multiple types of network functions with large numbers of instances, and evolutions on a continuous basis. We design and implement CORNET (Section 3) to tackle the challenges faced by change management operations. We conduct thorough evaluation (Section 4) of CORNET using real-world data collected from a large operational 4G and 5G cellular network and virtualized services such as VPN and SDWAN running in the cloud as well as experiments on a testbed of virtualized network functions. We share our experiences (Section 5) through use of CORNET within a large service provider over last three years to manage more than 7 million changes. CORNET has resulted in significant improvement in operational efficiencies, and faster and safer change deployments.

## 2 BACKGROUND AND MOTIVATION

In this section, we describe a typical change management flow, challenges faced in large network operations and highlight the motivation behind composition.

### 2.1 Change management flow

Any network function can undergo changes. Our interactions with multiple operations teams indicate that they all share a common change management (CM) flow. A change activity starts from designing the change that involves packaging the different components via an upgrade, feature activations, configuration changes, and hardware updates.

The process of executing a change is captured using a method of procedure (MOP) that details a sequence of steps such as pre-checks, configuration snapshot, the change implementation, and post-checks. The roll out is done in stages. The network engineering teams start by rigorously testing the change design and the MOP in a laboratory environment. Then the change is trialed on a small

part of the production network (typically referred to as the First Field Application – FFA). A pre/post comparison of network and service performance metrics, failures, and alarms is conducted to make a go/no-go decision for a network-wide deployment.

During the network-wide deployment, the idea is to quickly roll-out the change across all the instances in the network. The operations teams continue to monitor the network and service performance. If there is any unexpected performance degradation or increase in failures, then a decision is made to halt the roll-out to the rest of the network. Depending on the root-cause analysis and mitigation strategies, they either roll-back the change or patch the network through appropriate changes.

### 2.2 Challenges of change management

In this section, we highlight using real-world data as well as several operations teams interviews why the problem is important and challenging. We analyzed network changes using data collected over three years from a large service provider network.

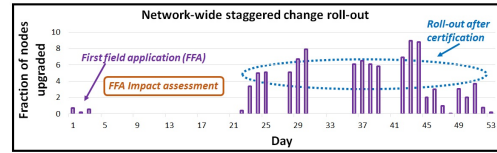
**Scale and complexity.** We observed that majority of the changes are executed in the edge of the network as compared to the core and the transport network. This is because the number of nodes in the edge are typically two orders of magnitude larger than transport and core. The changes are also frequent. For our network with nodes on the order of hundreds of thousands, we observe that for each day, the number of change activities is 10-20% of the number of nodes.

Table 1 shows the distribution of changes, the average change duration per node in terms of maintenance windows (a maintenance window is a time-slot, typically, mid-night to 6 AM local time, for implementing the change activities), and the average network-wide roll-out time for software upgrades, configuration changes, node re-tuning, and construction works. The node re-tuning change activity consists of the spectrum work done to carve out carrier frequencies from one technology (e.g., LTE) and re-provision for the new technology (e.g., 5G). The average change duration is higher for node re-tuning and construction works, because of the nature of the work activity and reliance on humans to physically go out to the site to conduct the work. The key takeaway from Table 1 is the relative magnitude of time taken for different types of change activities and so, it is agnostic to the actual unit.

**Table 1: Change distribution, average duration per node, and average network-wide roll-out time (60K+ nodes) per change type in maintenance windows.**

	Change activities	Avg. duration per node	Avg. roll-out (60K+ nodes)
Software upgrade	24.67%	1.92	63
Config. change	65.82%	1.66	35
Node re-tuning	1.14%	3.82	continuous
Construction work	8.37%	3.01	continuous

The sheer scale of the number of nodes makes it challenging to roll-out the change across the whole network. It is standard practice to stagger the roll-out to minimize the service disruption during the deployment and minimize the risk of any unexpected service impact after the deployment. Fig. 1 shows an example staggered software upgrade deployment across the whole network for 4G eNodeBs. The FFA deployment is a few days with the assessment of the impact spanning multiple days, and then after certification of



**Figure 1: Network-wide staggered deployment.**

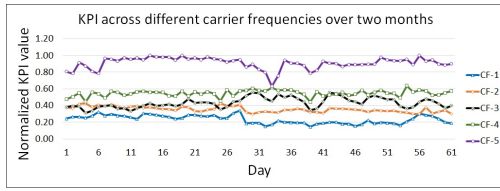
the change, large-scale network-wide roll-out spans several days. If the change is service disruptive (e.g., requiring a node reboot), then we do not want to change too many nodes simultaneously, thus minimizing the risk of a significant reduction in network capacity that, in turn, can negatively impact service performance.

We observe that large provider networks have on the order of hundreds of different types of nodes. Even though the scale is large at the edge, core nodes have much larger physical and logical connectivity which creates lots of dependencies and complicates planning of any change activity on them.

**Risk of service impact.** The change deployment needs to be carefully planned to ensure software compatibility across different network function types. Virtualization can further influence the risk of service impact. A simultaneous change activity on a primary virtual network function (VNF) and the physical server hosting the backup virtual network function can render both primary and backup VNFs unavailable. Thus, it is imperative to capture the physical server to VNF cross-layer dependency when planning their changes. We also need to consider service-layer dependency (e.g., a service chain consisting of a sequence of nodes). Multiple nodes being brought down within a single service chain simultaneously can help speed up the change deployment (assuming other service chains are available to the end-users) but introduces significant challenges for troubleshooting and localization during degradations. Redundancy in the network design, while a very beneficial concept, also makes troubleshooting harder for operations teams (e.g., figuring out how traffic got re-routed).

After the change is deployed, it is important to compare the health and performance before and after the change. Depending on the service, the operations team will measure multiple key performance indicators (KPIs). For example, in 4G/5G cellular service, the operations teams monitor several KPIs such as successful voice and data connection establishments, data throughput, call drops, and latency. A KPI is typically defined using multiple performance counters. For example, there are multiple counters to capture the reasons behind voice call drops (also referred to as **cause code** for drops) such as poor reference signals, radio link failures due to retransmission timeouts, handover execution failures, carrier failures. Different network changes can result in different performance impacts, and they can have different propagation behaviors.

**Diversity.** There is significant diversity in configuration across the network. As an example from 4G networks, we have 27 different radio head types, 13 carrier types (e.g., serving a network designed for first responders [7] or narrowband internet of things - NBIoT), and 5 downlink MIMO modes (for multi-path wireless communication). We use real-world performance and configuration data to show that diverse configuration exhibits different service performance. As an illustrative example in Fig. 2, data throughput on 4G eNodeB varies across different carrier frequencies (CF-1 to 5). Higher carrier



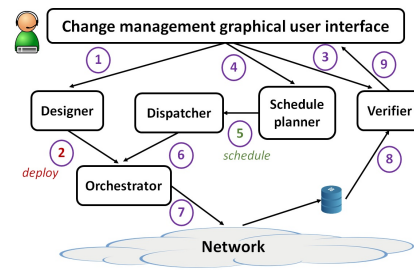
**Figure 2: Diverse KPI values across different carrier frequencies (CF-1 to 5).**

frequencies (CF-5) offer better data throughput compared to (CF-1). Lower carrier frequencies provide better coverage and can cover far-away users from the eNodeB. Now, observe in Fig. 2 that on day 28, there is an upward level change (improvement in data throughput) for CF-3, but downward level change (degradation) for CF-1 and CF-2. Thus, if one were looking at the KPI combined across CF 1 to 5, one may miss out on the interesting impact at individual carrier frequencies. The operations experiences also shed light that, often at times, the FFA (First Field Application) change trials can show the expected performance impacts, but network-wide roll-out can show unexpected impacts. This happens due to the diverse configuration and previously unknown interactions between them. But it also means that when rolling out changes, we may have to group nodes based on different configurations to simplify post-change analysis of performance. Our interactions with the network operations teams also depicted that different changes can have different scheduling requirements. We also requested MOPs from operations teams across multiple network function types and services to analyze how similar or different they are. Different network functions, vendors, hardware versions, and even sometimes, software releases have significant differences in the commands within their MOPs. Today, we lack standardization across network functions, and thus, the operations teams face the daunting task of creating, testing, managing, and implementing MOPs.

**Evolution.** Networks are continuously evolving in the form of virtualization and containerization of network functions, the introduction of new technologies and services such as 5G, software upgrades, and configuration changes to enhance service experience and fix network bugs. This continuous evolution makes change management an ongoing challenge for the network operation. New software dependencies need to be modeled when scheduling changes (e.g., 4G eNodeB and 5G gNodeB have to be concurrently upgraded to ensure software compatibility and seamless handovers for end-users). Key performance indicators (KPI) need to be updated whenever there is a change in the underlying counters across software releases (e.g., new failure cause code for voice calls introduced with the new software version). If the new cause codes are not accounted for during the network change roll-out, then any degradations caused by the new codes would not be captured in the pre/post-impact comparisons. Thus, it is vital to be agile in updating the different compositions across changes.

### 3 CORNET DESIGN AND IMPLEMENTATION

In this section, we present the design and implementation of CORNET. Fig. 3 shows the different components and a unified user experience to simplify the tasks in change management.



**Figure 3: CORNET overview.**

#### 3.1 Modular and re-usable design

In order to support high re-use and flexible composition across different network functions, we propose modular decomposition of the change method of procedure into reusable building blocks across all phases of change management. Each building block (BB) or software module is defined using an input/output parameter list, and has a REST API. Its meta-data (API location, input/output parameter definitions) is stored in our catalog. The input/output parameters for each building block have to be carefully defined in order to support composition or stitching of the building blocks into a workflow. The challenge is to determine how many building blocks to create using a change MOP with the appropriate degree of re-usability. Too big a building block would make it hard to re-use across different change activities. However, too small a building block would make it hard to compose the change workflow because one has to ensure proper propagation of parameter values across building blocks in the change design.

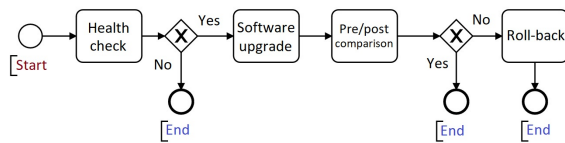
After surveying several change MOPs and discussions with operations teams across multiple services, we identify the common building blocks or modules as shown in Table 2. For example, a traffic redirect building block can be re-used across change MOPs for either software upgrades or configuration changes. We introduce a new concept of network function (NF) agnostic building blocks. The idea is to define and implement the building block capability that is independent of any network function and thus can be re-used across all functions. Ideally, we would like to make all building blocks NF-agnostic. However, due to lack of standardization across vendors, certain building blocks like traffic redirect or software upgrade require vendor or network function specific implementation. Our NF-agnostic BBs such as optimization solver, KPI aggregation, or impact detection are typically data analytic capabilities, and apply to any network function. We also design and implement these NF-agnostic BBs such that they can support flexible composition across different attributes such as optimization constraints, location attributes, and performance metrics.

#### 3.2 Change workflow designer

We propose a graph-based approach to design the change workflow (WF) using the building blocks (BB). The building blocks serve as the nodes and the connections between pairs of blocks serve as the edges of the graph. The implementation of the building block can use any programmable interface such as Ansible [12] playbooks, NetConf [45], Chef [20] recipes, Python scripts, or vendor-specific CLIs (command line interface). We use BPMN [4, 8] (business process modeling and notation) to implement our graph-based change workflow designer. We show a simple software upgrade workflow

**Table 2: Building blocks in CORNET’s catalog.**

Phase	Building block	Function	NF-agnostic
Design and orchestration	Health check	Verify live and operational status	✗
	Conflict check	Ensure no conflicting activities	✓
	Traffic redirect	Migrate traffic away before the change	✗
	Software upgrade	Implementation of the upgrade	✗
	Config change	Implementation of the config change	✗
	Pre/post comparison	Compare before and after the change	✓
	Traffic restore	Bring traffic back after the change	✗
Schedule planning	Roll-back	Restore to the previous version	✗
	Detect conflicts	Identify conflicting changes	✓
	Extract topology	Identify dependent nodes	✓
	Extract inventory	Identify attributes for constraints	✗
	Model translation	Intent to low-level constraint templates	✓
Impact verification	Optimization solver	Discover schedule	✓
	Change scope	Identify scope of change	✓
	Extract KPI	Collect data for pre/post	✗
	Extract topology	Identify nodes for relative comparison	✓
	Extract inventory	Identify attributes for aggregation	✗
	Aggregate KPI	Aggregate across attributes	✓
	Impact detection	Statistical comparison of KPI	✓

**Figure 4: An example software upgrade workflow.**

example using BPMN in Fig. 4 with four building blocks - health check, software upgrade, pre/post comparison, roll-back - and two decision operators after health check and pre/post comparison. If the health check fails (decision output is no), then the action is to end the workflow. If the pre/post comparison after the software upgrade fails (decision output is no), then the action is to roll-back the software. The input to the workflow (or, start) is the node and the software version for its upgrade. The output of the workflow is the status of execution. Some of the building blocks can generate new output variables that can be input for others. We capture the variables using global state information within the graph. The change workflow also has input and output parameters that have to align with the inputs and outputs across the building blocks. Before CORNET, the change workflows were specifically designed and implemented for each network function and composition. We design the workflows to be also NF-agnostic so they can be re-used across any network function. This code re-use brings in a tremendous degree of time savings for the network operations teams when designing their changes workflows (or MOPs). Our designer still allows the quick and flexible creation of any new workflow based on available building blocks.

After the design of the change workflow is complete, we propose a verification step where we ensure that there are no zombie building blocks (*i.e.*, no incoming or outgoing edge to another building block or decision block or start/end). We propose and implement a simple approach to discover such zombie building blocks by verifying that each building block within the workflow has an incoming and outgoing edge to other blocks. A change workflow without any zombie is deemed verified and we can then get ready for deployment into the orchestrator. For workflow deployment, we take the BPMN graphical layout with building blocks captured using the corresponding REST APIs and then dynamically create the WAR (web application resource) file which is the meta-code stitching of the different building blocks into a workflow. Different change activities can require different WAR files and having CORNET automatically generate the WAR enables the operations teams to easily

and dynamically generate different change workflow compositions. For example, in some scenarios, if operations teams would like to stitch the software upgrade followed by the configuration change on the same or co-located node, our change workflow designer can easily support such compositions using the graphical layouts. The WAR can then be referenced using a dynamically generated REST API for the newly created change workflow. For each workflow created within our change designer, we associate it with the corresponding WAR and REST API. The REST API information is important during the run-time dispatching and invocation of the change execution.

**Remarks.** An alternate design strategy to workflow-based change composition is to use event-driven (or, policy-based) composition of changes where building blocks are invoked based on events triggered by other building blocks. The communication between building blocks in event-driven approach happens via messages versus workflow-based approach uses explicit invocation of a REST API. We argue that change design, state management, and fall-out troubleshooting is easier with workflow-based approach. In the future, we plan to quantitatively compare the approaches.

### 3.3 Change schedule planner

The goal of the change schedule planner is to identify the schedule to deploy the changes across the network. The objective is to minimize the total completion time for the deployment subject to operational and service impact constraints. The operational constraints aim to align restrictions with respect to the change deployment such as the number of concurrent change executions permitted, or interleaving of changes across different nodes. The service impact constraints aim to minimize the risk of service disruption during and after the change deployment. Different change work-groups and services have different change scheduling constraints based on the network function the group is scheduling. Given that we have on the order of hundreds of network functions, designing custom algorithms or mathematical models (through standard mixed-integer programming approaches) for each network function is a very onerous task. Therefore, we take a dynamic, on-the-fly model-driven approach to formulate the change schedule planning problem. Based on the change deployment intent, constraints, and optimization objective, we automatically generate the mathematical models. We call this dynamic and on-the-fly because each model can vary based on the intents and attributes across different types of change activities. We capture the operational intent using high-level constraint rules. Depending on the composition of the constraint rules, we automatically map the constraint rules to constraint templates using MiniZinc [43], which is an open-source constraint modeling language to create the mathematical models. For each MiniZinc model that is dynamically generated based on operational intent, we use the optimization solver (constraint programming solver such as Google OR-Tools [47], or mixed-integer linear programming solver such as COIN-OR CBC [21]) to identify the best schedule conforming to the constraints. Note that the operations teams only deal with high-level scheduling constraints rules (or intent) and do not need to understand or modify the underlying constraint templates.

3.3.1 *Capturing scheduling intent and constraint templates.* Through several discussions with operations teams, we learn that the constraints for the change schedule planning can be associated with the following templates.

**Conflict scope constraint** aims to avoid concurrent change activities on the same network function instance by multiple work groups, or dependent instances on the service chain (e.g., conflict between vGW and its hosting physical server in SDWAN).

**Concurrency constraint** limits the number of network function instances to be executed concurrently within a group (e.g., Element Management System - EMS, or pool). The operational intent restricts concurrent executions to ensure either network availability, or conform to any execution limits.

**Consistency constraint** tries to schedule dependent changes together. For example, co-located 4G and 5G radio software upgrades have to be deployed close in time to ensure software compatibility.

**Uniformity constraint** tries to schedule changes such that instances have the same attribute values. For example, schedule instances together that belong to the same time-zone (or, sometimes nearby time-zones).

**Localize constraint** tries to complete all instances within a group before starting the next. This is to ensure smooth change deployment and simplified impact verification.

**Conflict tolerance constraint** tries to discover the change schedule plan that is either conflict-free (zero tolerance) or minimal conflicts. Most of the times, the operations teams aim to request conflict-free schedules. However, there are times during emergency deployment of changes or those that do not result in any service disruption (e.g., certain configuration changes), one would want to have a tighter schedule with minimal conflicts.

3.3.2 *Constraint composition and plan optimization.* We allow the network operations teams to pick-and-choose (multiple) instances of high-level change plan intents. The dynamic composition of scheduling constraints refers to the selection of different combinations of constraint templates and attributes. For example, one operations team may choose the composition of concurrency, localize, and conflict constraint for LTE eNodeB change schedule discovery, versus other may choose the composition of concurrency, consistency, uniformity and conflict constraint for the 5G gNodeBs. CORNET translates these high-level change plan intents into low-level mathematical constraints modeled in MiniZinc.

Such composition and the subsequent translation is not straightforward and presents several interesting challenges. In general, the constraints and attributes may have inter-dependencies. So, besides translating individual constraints, we also need to introduce some 'linking variables' to ensure correctness. A simple example is a node with two attributes of a market and a hardware version cannot be scheduled independently. Thus, if we schedule a market on a specific time-slot, it implies that nodes belonging to that market with potentially different hardware versions can be scheduled on the same time-slot. So, we need to add additional variables and constraints to model those dependencies.

We propose four aspects to accomplish composition with change planning: 1) Elementary Schedulable Attribute (ESA), which defines the atomic unit to be scheduled, 2) Conflict Attribute (CA),

which defines which attribute should be considered when checking conflicts (can be same instance or neighbors); 3) time granularity, which defines discrete time-slot where ESA can be scheduled; and 4) the dynamic set of constraints. Note that these definitions are specified according to the supplied inventory, and therefore, they are agnostic to the element type.

There are many ways of translating constraints and creating a final model and associated variables. The choice of which model to create has implications on scalability of CORNET. As an example, consider an operator who wants to schedule with the constraints that, on any given maintenance window, we schedule at most 300 nodes, belonging to at most 3 different markets, and no more than 150 nodes from any given market. They choose three high-level intents (a) concurrency constraint on the ESA attribute `common_id` (which is the network function instance), (b) concurrency constraint on `common_id` concerning to the market that is derived from the inventory attributes, and (c) concurrency constraint with an aggregate attribute. Also assume that the time granularity is one day. The following code fragment expresses such intents.

```

1  [{
2    "name": "concurrency",
3    "base_attribute": "common_id",
4    "operator": "<=",
5    "granularity": {"metric": "day", "value": 1},
6    "default_capacity": 300
7  }, {
8    "name": "concurrency",
9    "base_attribute": "market",
10   "operator": "<=",
11   "granularity": {"metric": "day", "value": 1},
12   "default_capacity": 3
13  }, {
14   "name": "concurrency",
15   "base_attribute": "common_id",
16   "aggregate_attribute": "market",
17   "operator": "<=",
18   "granularity": {"metric": "day", "value": 1},
19   "default_capacity": 150
20  }]

```

The first decision point for CORNET is the set of variables to model. The first concurrency constraint is relatively straightforward. We can use binary variable  $x_{it} \in \{0, 1\}$ , for all  $i \in E$  and  $t \in T$ , where  $E$  is the set of `common_ids`, and  $T$  is the set of available timeslots for scheduling.  $x_{it} = 1$  indicates that `common_id`  $i$  is scheduled on timeslot  $t$  and the first concurrency constraint can be expressed as

$$\sum_{i \in E} x_{it} \leq 300, \forall t \in T \quad (1)$$

The concurrency constraint on the market level is trickier because the attribute `market` is non-ESA. Therefore, CORNET must figure out the mapping between the ESA `common_id` and the non-ESA market. The way such data structures are built depends on the whole set of constraints. In general, we favor sparse representations, which are more efficient for solving such constraint programming or mixed-integer programming models. However, some constraints are better expressed with dense matrix representations.

Another decision that must be made is whether new decision variables for market should be created. Here, we have a performance versus expressiveness trade-off. The creation of new decision variables adversely affects solver performance especially when the distinct instances of non-ESA attribute (here `market`) are close in number to distinct instances of ESA (here `common_id`). On the other hand, such new variables make the modeling simpler, and we can reuse such variables across several constraints. Therefore, assume the decision variable  $y_{mt} \in \{0, 1\}$ , for all  $m \in M$

and  $t \in T$ , where  $M$  is the set of markets and  $y_{mt} = 1$  indicates that market  $m$  is scheduled on timeslot  $t$ . Let  $Q = \{(i, m) : \forall i \in E, m \in M \text{ such that } i \text{ belongs to } m\}$ , be the sparse mapping between `common_ids` and markets. We may model as

$$y_{mt} \geq x_{it}, \forall (i, m) \in Q, t \in T. \quad (2)$$

$$\sum_{m \in M} y_{mt} \leq 3, \forall t \in T. \quad (3)$$

Note that the actual concurrency constraint is Inequality (3). Inequality (2) is a link constraint necessary to tie the `common_id` with the market. So, we not only create extra decision variables but also add a new set of constraints in the model.

If we do not create a decision variable  $y$ , we can express the second concurrency constraint only in terms of the ESA:

$$\sum_{m \in M} \sum_{i:(i,j) \in Q} x_{it} \leq 3, \forall t \in T. \quad (4)$$

In this case, we reduce the number of variables by not creating decision variables to market and avoid the inclusion of many linking constraints. However, such representation is much denser than the first one and can make the problem harder to solve. Moreover, if we have multiple levels of dependency, e.g. `common_id`  $\rightarrow$  market  $\rightarrow$  region  $\rightarrow$  state, a unique constraint could be very challenging to represent and even harder to solve.

Finally, we can write the last concurrency constraint as

$$\sum_{i:(i,j) \in Q} x_{it} \leq 150, \forall m \in M, t \in T. \quad (5)$$

Note that we make use of  $Q$  again, and therefore, two different constraints have interdependency on the data. Note that if we have two non-ESA attributes in constraints like this, CORNET must evaluate whether it is worth creating decision variables and extra linking constraint or creating denser but lesser constraints.

Overall, the translation of high-level intent to low-level mathematical models is far from simple 1:1 mapping. There are other complications CORNET must deal with. For example, different time granularity among constraints, different units, attribute multiplicity, n:m attribute mappings, among others. CORNET must consider the whole model, identify common data structures and the best way to represent them, identify shared decision variables, and compute and compare the density of several alternative representation. Once done, CORNET must assemble the model in a very efficient way so that MiniZinc can process and send it to the solver. These MiniZinc models can be very long. Appendix B contains a full example. This intent to model translation is done automatically in CORNET, whereas in Zapper [22], the models were created manually and then available to be linked automatically with a user request.

**3.3.3 Scalable dynamic composition.** The optimization solvers in MiniZinc can effectively tackle a small number of network function instances. We will show in Section 4 that the MiniZinc solvers for composition across all six constraints can handle around 1,000 instances. This works nicely for core networks (typically, hundreds of routers or switches). However, it cannot tackle the large scale of operational networks close to the edge, which have on the order of hundreds of thousands of instances.

We propose new algorithms to scale up our dynamic composition for change schedule planning and optimization solvers in MiniZinc to tens of thousands of instances. Our two main ideas that can be

applied independently are (a) the consistency constraint forces us to schedule certain changes together. Thus, we could divide the changes into non-overlapping groups such that each group needs to be scheduled together and then solve our problem on the small number of groups. (b) we divide the changes into sets that have no dependencies with respect to constraints. Then, we can solve in parallel and combine their solutions.

The advantage of our dynamic composition solution is the flexibility across different scheduling requirements and network function types, but that comes at a price on the running time and the schedule quality. Based on the popular composition patterns across operations teams, we devise a few custom heuristic to deal with the large scale of the network. The composition in custom heuristic is “hard-coded” but the running time is much lower than the generic and flexible solver. It would be an interesting future research to bridge the gap between generic solvers supporting dynamic composition and faster discovery of change schedules. In Appendix C, we describe our custom heuristic to generate change schedules for tens of thousands of 4G eNodeBs and 5G gNodeBs.

### 3.4 Change dispatcher and orchestrator

After the change schedule plan across multiple network function instances is acknowledged by the operations teams, it is sent to the dispatcher along with the corresponding change workflow created using the designer. The dispatcher automatically invokes the change orchestrator at the specific time for the scheduled instances. If multiple instances have to be executed concurrently, the dispatcher handles that in run-time and as soon as workflows for individual instances complete, the workflow for the next instance is triggered. The change workflow execution is invoked by the orchestrator using the REST API information stored in the workflow meta-data, and the instance and change information passed by the dispatcher as input to the change workflow. We use Camunda [19] as our open-source change workflow orchestrator.

We enhanced the Camunda-based workflow orchestrator to automatically log the status of execution for each building block along with the time taken to complete its execution. If the change workflow completes through atleast one start to end flows, then the workflow is deemed as successfully completed. A failure to complete the workflow execution can be associated to the building block that failed to execute successfully. Our fine-grained logging thus enables the network operations teams to identify the offending building blocks based on their status of execution across multiple change workflows. Such post-hoc analysis of the workflow execution is often important to troubleshoot unsuccessful change executions. Furthermore, we incorporate pause and resume functionality within our orchestrator to support halts in the workflow, with each building block execution treated in an atomic fashion. This functionality enables operations teams to manually pause the automated workflow execution in case of any unexpected alarms or failures. After the operations teams troubleshoot and resolve the issue, they can resume the execution of the change workflow. We anticipate this pause and resume to be used rarely but whenever it needs to be used, it is of extreme value add. The pause and resume functionality was also crucial for change workflow execution that required manual interventions (e.g., hardware swaps).

### 3.5 Change impact verifier

As the changes are being executed by the orchestrator, the goal of our verifier is to compare performance and alarms before and after the change, and ensure the expected impacts of the change. We propose a new verifier that does a statistical pre/post impact comparison, supports composition across multiple impact metrics, and operates on multiple time-scales after the change to detect unexpected impacts. Multiple time-scales are important to account for the variability in the metrics. We can detect massive degradations on shorter time-scales such as minutes, or subtle impacts on longer time-scales such as days and then trigger actions such as roll-back of the change or halts in network-wide deployments.

The composition of the metric is important because the change intent can determine the nature of the impact. We enable the operations teams to create multiple verification rules for each change based on their expectation and the intent of the change. For example, a software upgrade can result in an expected improvement in voice call quality but a very minor degradation to data throughput, versus a subsequent configuration change can result in improvement to data throughput and latency. Incorporating the right metrics for the change is extremely important to verify its expected impacts and intent. By embedding these rules in our verifier, we can focus on the relevant impacts. Furthermore, in order to support composition of the metrics across different changes, we separate the data processing and rule generation from the statistical analysis.

**3.5.1 Verification rule composition.** We create multiple data adapters to support collecting data from multiple sources. We present a simplified interface to the operations teams agnostic of the underlying data to create their corresponding key performance indicators (KPI) and alarm queries. Our interface consists of the time during which the KPI have to be collected, their time granularity (minutes, hours, or days), network function instance identifier, the KPI equation/alarm tags. The KPI equations often change across major software releases and thus it is important for the operations teams to quickly modify them and incorporate into the rules for change impact verification. Through operations experience and interaction, we learn that the change impact can vary across different network locations based on morphologies (e.g., urban versus rural), and configuration attributes such as software or hardware versions. This can make it extremely hard to accurately verify the impacts across locations. We address this by supporting aggregation of the KPI across geographic locations and multiple configuration attributes. The aggregation function can be the average, median, or weighted average across the configuration attributes. We use the idea of study group (where the change was implemented) versus control group (change was not implemented) comparison to enable robust verification of the change impact. The control group selection criteria can also influence the change impact verification. We incorporate the network topology and inventory information to automatically derive the control group (e.g., first-hop neighbors with the same hardware version as the study group).

**3.5.2 Robust statistical analytics.** The goal of our statistical pre/post comparison is to identify if there is an improvement, degradation, or no impact of the change. We create a robust regression

model between the study group ( $S$ ) and control group ( $C$ ) KPI time-series for the interval before the change  $S = \beta C$ . Then, using the control group  $C'$  KPI time-series after the change, we predict the study group ( $\overline{S'}$ ) KPI time-series after the change. The intuition is the network change at the study group should induce the impact in the KPI time-series only in the study group and the control group should not be affected. If there is no statistical difference between the study group time-series prediction ( $\overline{S'}$ ) with study group measured after the change ( $S'$ ), then we can infer that there is no impact of the change. Our approach tackles uncorrelated effect of external factors that would influence both study as well as control groups.

We use a robust rank-order test of medians [26, 35, 40, 53] to compare  $\overline{S'}$  with  $S'$ . We repeat this analysis for each instance that undergoes the change. The next challenge is to account for the staggered roll-out of network changes. We tackle this through time-alignment and normalization analogous to Mercury [41] and support the automated summarization of the impact across multiple compositions (or, selections) of the KPI metrics in the verification rules and across multiple configuration attributes. The operations teams are then presented with the impacts for enabling a quick go/no-go decision to continue the roll-out or halt.

## 4 EVALUATION

We evaluate CORNET using experiments conducted on a testbed of virtualized network functions that mimics a production setup and by using real-world operational network data collected from a large service provider. We compare CORNET to custom solution in terms of (a) the degree of code re-use for supporting new network functions and new compositions, and (b) the quality of the solution.

### 4.1 Change designer and orchestrator

We collected two software images for each of the six sample virtual network functions (vNFs) from the operations teams managing the production cloud service for Virtual Private Network (virtual customer edge vCE router), Software defined Wide Area Network (virtual gateway for traffic tunneling from edge, portal for configuration and monitoring of SDWAN ecosystem, and customer premise network functions), and cellular virtualized core (vCOM for centralized operations management and vRAR for revenue assurance reporting) as part of the Voice over LTE service. More details about the three services can be found in Appendix A. The vNFs were instantiated using OpenStack. We created three building blocks – health check, software upgrade, and pre/post comparison.

Without CORNET, one would have designed and implemented a custom solution for the building blocks and the change workflow across each network function independently. With CORNET, we have the pre/post comparison building block and the change workflow implemented in an NF-agnostic way. These can be re-used across all network functions. The health check and software upgrade building blocks are specific to each network function and thus cannot be re-used across different functions. We used command-line scripts for vCE router and Ansible playbooks for the remaining five vNFs to implement health check and software upgrades. Thus, without CORNET, one would have to implement a total of 24 (18 NF-specific BB, and 6 NF-specific WF) modules, versus 14 (1 NF-agnostic BB, 12 NF-specific BB, 1 NF-agnostic WF) when using

CORNET for supporting change design and orchestration across six vNFs. This results in a code re-use of 42% due to CORNET. In order to test the quality of execution, we completed the software upgrade workflow execution for each of the instance separately and then verified that the software versions were successfully updated.

## 4.2 Change schedule planner

We collected change management logs (for planned changes across the network function instances), inventory information (hardware and software versions, timezones, EMS that the instances connected to, and the market they belong to), and topology data (connectivity between instances on a service chain). We used two radio access network functions (4G base station eNodeB, and 5G base station gNodeB), two transport switches, and two core network functions to evaluate the code re-use and the quality of change schedule output from CORNET. We tested dynamic composability exhaustively by considering all possible combinations of the following three constraints: consistency on the instances within an area, uniformity based on the same time-zone, and localize using the area. Then for each of the eight combinations, we considered two variations of conflict tolerance: zero conflict or minimal conflict, thus giving us a total of 16 combinations. We created five building blocks in our CORNET's catalog to support discovery of change schedule plan – detect conflicts, extract topology, extract inventory, model translation, and optimization solver.

Without CORNET, one would have to build custom solutions for each network function independently and supporting the 16 combinations of constraint compositions. This would result in one requiring to implement a total of 126 modules (30 NF-specific BB, 96 NF-specific WF). The 96 custom solvers are required to support each of the constraint composition and each network function. With CORNET, we implement four out of five building blocks and the schedule planning workflow in an NF-agnostic way and supporting any composition of constraints. This results in a significant code re-use requiring us to only implement 11 modules (6 NF-specific BB, 4 NF-agnostic BB and 1 NF-agnostic WF). This results in a code re-use of 91% due to CORNET.

For each of 16 combinations for constraint composition, we measured the change deployment time (time to complete the change execution across all input nodes, also commonly referred to as the makespan in the optimization community) and schedule discovery time (time taken by our change schedule planner to identify the deployment schedule). We always set the concurrency of 200 instances per EMS and the conflict scope of service chain. We fix the network function type of 4G eNodeB and vary the instances from 200 to 1,000 in increments of 200.

We make the following inferences based on our results. (a) Increase in the number of instances results in an increase in the schedule discovery time. (b) Some constraints (e.g., localize and uniformity) dramatically increase the schedule discovery time implying that our solver in CORNET has to work harder to converge to an optimal. The main reason is that such constraints are very dense in terms of the large number of new decision variables they introduce. Furthermore, these constraints induce the searching for permutations (or orderings) in the chosen attribute, which is exponential. (c) With the consistency constraint, we observed 4x

reduction in schedule discovery time because our model could exploit the consistency constraint to determine the schedule at the level of groups of nodes that need to be scheduled together and we have fewer instances of groups than the number of nodes, resulting in a smaller MiniZinc model.

Next, we evaluate the efficacy of the change schedule planning solution in CORNET and compare it to the custom solution (described in Appendix C) for a fixed set of constraints and scaling up to tens of thousands of nodes as input. The goal of this evaluation was to study the impact of our generic model-driven solver on the makespan and schedule discovery time. To scale up to a large number of nodes, recall that in CORNET (Section 3.3.3), we propose to add consistency constraint on an attribute derived using topology in addition to the constraints specified by the operations teams. Adding an extra constraint can potentially increase the makespan, but our results show that the makespan with CORNET increases by only around 7% compared to the custom solution.

## 4.3 Change impact verifier

We evaluated the flexible composition and re-use capability of our change impact verifier by creating multiple groups of key performance indicators (a total of 349 KPI equations provided by the operations teams) across different vendors, network function types, and multiple location aggregation attributes. We created six building blocks – identify scope of change, extract KPI data, topology, inventory, aggregate KPI across location aggregates, and detect performance impact. We created three compositions of attributes and impact verification rules across 3 network function types (4G eNodeB, 5G gNodeB, and a transport network switch). Without CORNET and using custom solution for each building block and workflow, one would have to implement 63 modules (54 NF-specific BB and 9 NF-specific WF). With CORNET, the number of modules needed to be implemented were only 11 (6 NF-specific BB, 4 NF-agnostic BB and 1 NF-agnostic WF), thus a code-re-use of 83%.

In order to verify the correctness of our change impact verifier, we asked the network operations teams to label the performance impact of the changes that were applied in a staggered fashion. The number of nodes involved in each of the change was on the order of tens of thousands. We had a total of 60 labels of impacts. Our change impact verifier in CORNET accurately identified all the 60 impacts as expected by the operations teams.

**Table 3: Code re-use improvements and loss in efficiency with CORNET as compared to custom solutions.**

	Code re-use	Loss in efficiency
Designer and orchestrator	42%	0
Schedule planner	91%	7%
Impact verifier	83%	0

**Summary.** CORNET achieves a high degree of code re-use compared to custom solutions in order to support multiple network function types and compositions. The higher the re-use, the less time one has to spend in realizing a change management solution. Thus, the re-use and composition capability of CORNET would bring in significant improvements to operational efficiency. However, we observe a minor loss in efficiency (e.g., 7% increase in makespan with CORNET's change schedule planner). We summarize these results in Table 3.

## 5 OPERATIONAL EXPERIENCES

In this section, we share our experiences from the deployment and use of CORNET in large scale operational networks for over three years. We demonstrate the benefit of re-use and flexible composition based on the usage patterns from network operations teams. We also show how CORNET is resulting in significant improvements in operational and network change deployment efficiency.

### 5.1 VPN and SDWAN software upgrades

The operations teams successfully used CORNET to design, plan, and orchestrate software upgrades across virtual customer (vCE) routers (approx. 1,000) in VPN network and virtual gateway and portal (approx. 50) in SDWAN.

**CORNET enables re-use, flexible composition and automated software upgrade deployment.** For vCE routers, the operations intent was to design and deploy changes using two workflows: one for software download and installation which is typically the time-consuming step and not service disruptive, and the second for health check, verification, reboot, and post checks to validate vCE and service availability. After completing the software installation across all vCE routers, they could plan the second workflow to be implemented a few days later. The building blocks and workflows were re-used across multiple software version upgrades. The schedule planning took into account that there were no conflicts with the changes on the underlying physical servers hosting the vCEs. The reboot for vCE took less than a couple of minutes and with some degree of parallelism across vCEs, the second workflow was executed for all vCEs four days after the first workflow. The impact verifier was then used across multiple performance metrics such as router CPU and memory utilizations and packet losses and discards. It was very interesting to verify the expected reduction in packet discard rates because of the new software deployment. The memory utilizations increased slightly because of the larger image size. This use case demonstrated how the operations teams leveraged the re-use capability and flexible composition in CORNET to deploy software upgrades across all vCE routers using a two-workflow approach. In a couple of occasions, we did notice failures of the software deployment. It was because of SSH connectivity issue and the fall-out at the time had to be dealt with manually. We learned through this experience that some failure modes can be outside the scope of initial design of the workflow and thus one has to have an “out of band access” to the vNF instances.

The virtual gateway and portal functions within SDWAN were individually upgraded using a single workflow (as opposed to two separate workflows for vCE). The workflow comprised of three building blocks pre-check, software upgrade with reboot and post-check. The building blocks were implemented using Ansible playbooks. CORNET’s workflow designer enabled the SDWAN operations teams to flexibly create the workflow and deploy for execution for both the network functions. The scheduling constraint was to ensure that the connected gateway and portal upgrades happened close in time to ensure software compatibility and conflicting changes with the underlying physical servers were avoided. The automated execution of the software upgrade enabled the network operations teams to reduce the work time from approximately 30

minutes to 4 minutes per instance. VPN and SDWAN software upgrades using the designer, dispatch and orchestrator of CORNET has been increasing the confidence across the network operations teams and helping with adoption for automated change execution across new network functions and services.

### 5.2 4G/5G cellular network changes

In this subsection, we share our results and insights from extremely high usage on a daily basis and adoption of our schedule planning and impact verification. The operations teams had been facing tremendous challenges with co-ordinating their change activities and verifying their impacts across tens of thousands of network function instances and hundreds of key performance indicators (KPIs). Over the last three years, CORNET has been used to schedule more than 7 million changes for LTE and 5G base stations (eNodeBs and gNodeBs respectively), and transport switches, and careful and continuous verification of the change impacts. On several occasions, CORNET identified subtle degradations in service performance that resulted in detailed root-cause analysis with vendors and resolution of the issues through subsequent software patches and configuration changes.

**CORNET supports flexible composition for change planning.** We observe the usage patterns across different change planning requests to infer that the network operations teams use several different compositions of the constraints. Our solver in CORNET can support different constraint compositions and different network element types. Different changes and different work groups have different schedule planning requirements. Minimizing the conflicts and ensuring the uniformity constraint to schedule nodes with the same timezone was the dominating pattern across a lot of operations teams because of their role in deploying configuration changes which are non-disruptive, and thus the goal was to quickly roll-out the change across the whole network. The next prominent usage pattern was across the operations teams deploying software upgrades with zero conflict tolerance, localize, uniformity and consistency constraints. The software upgrade team had to bring in the consistency constraint during the introduction of 5G for grouping the eNodeB and gNodeB upgrades together in order to support software compatibility and thus avoid any disruption due to handover failures. This shows the evolution of the constraints for the same team over time as new technology was introduced.

**CORNET improves operational efficiency in schedule planning.** Through interviews with the operations teams (approximately 30 work groups), we identified that before CORNET, the operations teams would take a batch approach where they would manually identify the schedule for a subset of nodes that are conflict-free and conforming to constraints. This task would take them around an hour sifting through databases and co-ordinating with other work groups, which was clearly tedious and time-consuming. They would then repeat this task until all nodes to be upgraded were covered. Now, with CORNET, the operations teams can request the change schedule across the whole network in a single request. For a network size of 100K, CORNET takes only a few minutes to automatically discover the schedule that conforms to multiple planning constraints. We used last three years of CORNET usage to track the change schedule discovery time. We compare this

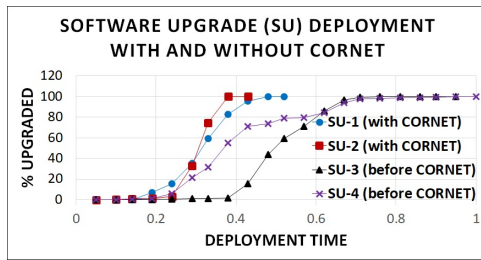


Figure 5: With CORNET, we observe a significant improvement in change deployment times.

time with the amount of time the operations teams were spending before CORNET. We calculate the average human time savings as the percentage reduction in schedule discovery time as a result of using CORNET. We observed that CORNET achieved an average human time savings of 88.6%, and thus a significant improvement in operational efficiency for change planning.

**CORNET planning achieves faster and efficient network-wide roll-out.** Fig. 5 provides an illustrative example<sup>1</sup> comparing the change deployment times for four software upgrades (SU) on eNodeBs - two planned using CORNET (SU-1 and SU-2) and two without (SU-3 and SU-4). The X-axis captures the change deployment time in slots. We normalize the values in X-axis by the maximum for proprietary reasons to not expose the total time taken for change deployment in production carrier networks. The Y-axis captures the percentage of nodes upgraded. As can be seen for all software upgrades, the initial time is spent in very slowly upgrading the nodes (FFA changes) and crawl-walk phase where the operations teams are carefully verifying the impact and making decisions for network-wide roll-out. In the run phase, the operations teams intent is to ramp up the deployment with continuous performance impact monitoring. With CORNET, the network operations teams are able to roll-out the change activity across the whole network much faster as compared to without CORNET. The reduction in the change deployment time is substantial and this enables end-users (or, service customers) to cherish the benefits of the network software upgrade faster. Also, we observe that the tail for SU-3 and SU-4 is very long as compared to SU-1 and SU-2. This indicates that CORNET is successfully able to discover the conflict-free schedules conforming to operational constraints, and thus the stragglers (nodes in the tail of the distribution for plan using CORNET) do not take a long time to finish. The change deployment plan is thus quite compact when using CORNET. This is possible because CORNET's schedule planner with a global view upgraded some of these straggler nodes earlier in the deployment. This result strongly highlights the benefit of using CORNET in improving the impact to network, and accomplishing faster roll-out of changes network-wide.

**CORNET supports flexible composition of impact verification rules.** We observed significant number of different compositions across different changes and work groups. A large number of impact verification queries had a small number of KPIs (< 20) belonging to the scorecard group. This was the typical pattern for

<sup>1</sup>The operations teams feedback has been that with CORNET, they have been seeing a tremendous improvement in network-wide roll-out plans for several software releases and configuration changes. Due to space restrictions, we only provide 4 examples (2 with CORNET and 2 without).

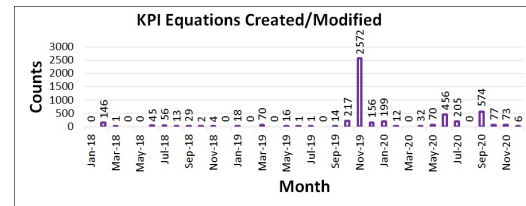


Figure 6: KPIs created or modified by the operations teams. Significant increase since September 2019 in preparation for 5G service roll-out.

a network-wide roll-out. A small number of queries comprised of a large number of KPIs (between 100 and 400). This was the typical pattern from users conducting FFA impact verification. As per the operations teams, FFA impact verification has to be rigorous in order to make a go/no-go decision and then experience a smooth expected roll-out. In Fig. 6, we capture the evolution over last three years of the KPI definitions created or modified by the operation teams. As can be seen, the KPIs are continuously added or modified on a monthly basis. We observe a significant increase since September 2019 which is to prepare for impact verification of 5G service roll-out. We learned that with new technology introduction (5G), we need CORNET to be extremely agile to support dynamic adaptations of KPI definitions and verification tests around 5G software upgrades and configuration changes.

**CORNET enables rigorous impact verification and safe network-wide roll-out.** Table 4 summarizes the use of CORNET in continuously verifying the performance impact of software upgrades and configuration changes on 4G eNodeBs and 5G gNodeBs every year over the last three years. As one can see, the FFA trial is typically conducted over hundreds of nodes. Only about 10% of the FFA trials are certified for a network-wide roll-out. The roll-out is across tens of thousands of nodes in a staggered fashion typically spanning around two months. The operations teams use CORNET to automatically monitor a large number of KPIs and across a large number of location aggregation attributes as the changes are being rolled out in a staggered fashion. This results in a significant time reduction (approximately 98%) in verifying the impact of network changes, thereby leading to a significant improvement in operational efficiency! Study versus control comparison also ensures a robust impact assessment. Any impact detected by CORNET is notified to the operations teams that use the rules to make a go/no-go decision. Our experience has been that sometimes even subtle degradations can result in a halt. Root-cause analysis and troubleshooting is then conducted (often with vendor help) and then decision is made to either roll-back the change or fix using a patch. The schedules are then adjusted accordingly using CORNET's schedule planner.

Our location attribute based aggregation of impact actually turned to be very positive functionality because it highlights which attribute or network configuration is contributing to the unexpected performance impact. This information was then used in a few occasions to halt only for a few specific nodes with the problem configuration instead of halting across the whole network. Thus, the rest of the network could still be upgraded while the patch was being developed for the problematic configuration. This on-the-fly and optimized roll-out process enabled intelligent roll-out strategy

**Table 4: CORNET use in verifying the impact in 4G and 5G cellular networks every year over the last three years.**

Change type	Number of FFA	Nodes per FFA	Number of certified roll-out	Nodes per roll-out	Number of certified roll-outs that got rolled back
Software upgrade	~ 160	O(100)	~ 16	O(10K)	< 2
Configuration change	~ 200	O(100)	~ 20	O(10K)	< 2

for the operations teams. Operations teams also shared their experiences over multiple years that by hardening the FFA impact verification, they have been able to make the network-wide roll-out much smoother and eliminate any unexpected performance impact during the roll-out, thereby reducing the halts or need to roll-back. Given the hardened FFA verification, they still like to use CORNET to verify the impact during the roll-out as a safety measure.

### 5.3 Lessons Learned

**CORNET deployment and adoption.** In the early phases of CORNET design and deployment, we went through several iterations to converge on the key aspects of change management that needed simplification to provide the most benefits to the operations team. For example, capturing the high level intent for change schedule discovery across different teams from their day-to-day practices was a challenge. Getting a buy-in from operators on a fully automated solution was hard. They were understandably hesitant to place their faith in a system that they did not completely understand. So we created intermediate solutions to bridge the adoption. For example, operators would guess a manual solution and then check with their peers in other groups for possible conflict so we automated that part of conflict checking. As they got comfortable, they slowly transitioned to automated schedule discovery. We also found that sometimes operators were conflating what was a true operational constraint and what were artificial constraints they had imposed in order to simplify manual discovery of solutions. Over time, this distinction blurred and it took us some time to tease out the minimal set of constraints that were needed.

The composition and re-use aspect was indeed transformational for the operations teams and it did take some time for them to adapt to the new guidelines and start to appreciate its value. Creation of re-usable building blocks and workflows, generic constraint templates and KPI equations required time investment from the operations teams. Soon, multiple teams observed the value of re-use and sharing and then it was a win-win situation for all. Simplifying the result presentation with sufficient diagnostic information is also key to easy interpretation and quick adoption.

**CORNET usage and maintenance.** Given the high usage of CORNET and reliance by the network operations teams for crucial decision making to deploy and manage network changes, it is extremely important to quickly account for the data integrity issues that arise in production networks. The data is key to accurate analysis and inferences and thus any delays, missing measurements and incorrectness can cause significant overload and distress to the operations teams. Over time, we enhanced our analytics to be robust to missing or inaccurate data and put in place regular monitoring of data feeds to detect and alert issues. As an example, in 4G cellular networks, we know that there is a common switch to all co-located eNodeBs so even if some of the eNodeB-switch relationships are inconsistent, we can infer correct connections based on taking a union of last five days' worth of data. This takes care of majority of

missing topology data in day to day snapshots. The downside is that if there is an actual decommission from the network, we will still see in our topology and this may force an unnecessary dependency, but the final effect is to make our schedules more conservative.

## 6 LIMITATIONS AND FUTURE WORKS

**High-level intent completeness.** Our goal in CORNET is to devise a set of intents that closely mirror how operators think of constraints to make it easy for them to use our system. The composition capability in CORNET is designed such that we can easily integrate any new intent without re-writing others which might be needed in custom solutions. However, for new intents, it takes some level of mathematical sophistication to translate network operator's intent (expressed in natural language) to our mathematical intent and guarantee that they indeed capture network operators' intent. This remains an area of future exploration. How do we provide a useful set of high levels intents that are (a) easiest for network operators to use, (b) efficient to use in our system and (c) cover a large set of constraints? There are clear trade-offs because providing a set that is complete but low level may not be useful for the operators and in some cases, we may not be able to translate them efficiently to pass to solvers.

**End-to-end automation.** An open question is how far can we push the change management operations with respect to end-to-end automation and decision making. In CORNET, we demonstrated composition and automation in each component of the change management flow. However, there are still touch-points where manual input is key. For example, providing high-level intent for change schedule planning or a decision for roll-back based on very subtle degradations. The risk to complete automation is unnecessarily halts with false alarms, slow roll-outs with conservative planning, or delays in alerting if automation went berserk.

## 7 CONCLUSIONS

Change management is an important problem for network operations. In this paper, we highlighted the need for quick and easy adaptation of its capabilities to keep up with the continuously evolving networks. We proposed a new framework, CORNET, that automatically translates high-level intent into low-level implementation. It incorporates modular composition using re-usable building blocks, automated translation of change planning intent into mathematical constraints and models, invoking optimization solvers to discover schedules, and robust verification of change impact. We evaluate CORNET using real-world data collected from operational networks and experiments on a testbed of virtualized network functions. Our operational experiences from using CORNET over the last three years demonstrate the importance of composition in change management functions. CORNET is resulting in significant improvements in operational and network change deployment efficiency.

## Acknowledgment

We thank our shepherd Arpit Gupta, ACM SIGCOMM anonymous reviewers, Zihui Ge, Vijay Gopalakrishnan, Jennifer Yates, and Robert Riding for their insightful feedback on our paper. We commend the collaboration and the support from the Network Engineering and Operations teams in the application and use of CORNET over several years in operational networks. We also thank Vikas Varma, Avteet Chayal, Subhash Kapoor, Kay Adcock, Craig Scantlin, Joseph Kochinski and Jie Wang for some of the evaluation of CORNET and implementation and deployment in production environments.

## REFERENCES

- [1] 2020. AT&T provides some details about March 5 FirstNet data-service outage. Retrieved January 26, 2021 from <https://urgentcomm.com/2020/03/18/att-provides-some-details-about-march-5-firstnet-data-service-outage/>
- [2] 2020. The great 2020 Gmail outage: A tale of two blackouts, and lessons learned. Retrieved January 26, 2021 from <https://www.itworldcanada.com/article/the-great-2020-gmail-outage-a-tale-of-two-blackouts-and-lessons-learned/439924>
- [3] 2020. T-Mobile screwups caused nationwide outage, but FCC isn't punishing carrier. Retrieved January 26, 2021 from <https://arstechnica.com/tech-policy/2020/10/fcc-not-punishing-t-mobile-for-outage-that-ajit-pai-called-unacceptable/>
- [4] 2021. BPMN IO Viewer and Editor. Retrieved January 26, 2021 from <https://bpmn.io/>
- [5] 2021. Facebook says 'configuration change' caused some users to be logged out unexpectedly. Retrieved January 26, 2021 from <https://www.theverge.com/2021/1/23/22245842/facebook-logged-out-configuration-change-ios-app-security>
- [6] 2021. Fastly blames software bug for major global internet outage. Retrieved June 14, 2021 from <https://www.reuters.com/business/media-telecom/fastly-blames-software-bug-major-global-internet-outage-2021-06-09/>
- [7] 2021. Firstnet. Retrieved January 26, 2021 from <https://www.firstnet.com/>
- [8] 2021. Object Management Group Business Process Model and Notation. Retrieved January 26, 2021 from <https://www.bpmn.org/>
- [9] 2021. Summary of the Amazon S3 Service Disruption in the Northern Virginia (US-EAST-1) Region. Retrieved January 26, 2021 from <https://aws.amazon.com/message/41926/>
- [10] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. 2020. AED: Incrementally Synthesizing Policy-Compliant and Manageable Configurations. In *Proceedings of the 16th International Conference on Emerging Networking Experiments and Technologies*. Association for Computing Machinery, New York, NY, USA, 482–495. <https://doi.org/10.1145/3386367.3431304>
- [11] Omid Alipourfard, Jiaqi Gao, Jeremie Koenig, Chris Harshaw, Amin Vahdat, and Minlan Yu. 2019. Risk Based Planning of Network Changes in Evolving Data Centers. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) (SOSP'19). Association for Computing Machinery, New York, NY, USA, 414–429. <https://doi.org/10.1145/3341301.3359664>
- [12] Ansible. 2021. Drive automation across open hybrid cloud deployments. Retrieved January 26, 2021 from <https://www.ansible.com/>
- [13] Mona Attariyan, Michael Chow, and Jason Flinn. 2012. X-ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX Association, 307–320.
- [14] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A General Approach to Network Configuration Verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (Los Angeles, CA, USA) (SIGCOMM '17). Association for Computing Machinery, New York, NY, USA, 155–168. <https://doi.org/10.1145/3098822.3098834>
- [15] Ryan Beckett and Ratul Mahajan. 2020. A General Framework for Compositional Network Modeling. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks* (Virtual Event, USA) (HotNets '20). Association for Computing Machinery, New York, NY, USA, 8–15. <https://doi.org/10.1145/3422604.3425930>
- [16] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. 2016. Don't Mind the Gap: Bridging Network-Wide Objectives and Device-Level Configurations. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) (SIGCOMM '16). Association for Computing Machinery, New York, NY, USA, 328–341. <https://doi.org/10.1145/2934872.2934909>
- [17] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. 2017. Network Configuration Synthesis with Abstract Topologies. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) (PLDI 2017). Association for Computing Machinery, New York, NY, USA, 437–451. <https://doi.org/10.1145/3062341.3062367>
- [18] Rüdiger Birkner, Dana Drachler-Cohen, Laurent Vanbever, and Martin Vechev. 2020. Config2Spec: Mining Network Specifications from Network Configurations. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA.
- [19] Camunda. 2021. Camunda Microservices Orchestration. Retrieved January 26, 2021 from <https://camunda.com/solutions/microservices-orchestration/>
- [20] Chef. 2021. Chef. Retrieved January 26, 2021 from <https://www.chef.io/>
- [21] COIN-OR. 2021. Cbc: COIN-OR branch and cut. <https://doi.org/10.5281/zenodo.3700700>
- [22] Carlos Eduardo de Andrade, Ajay A Mahimkar, Rakesh K Sinha, Weiyi Zhang, Andre Cire, Giritharan Rana, Zihui Ge, Sarat Puthenpura, Jennifer Yates, and Robert Riding. 2021. Minimizing Effort and Risk with Network Change Deployment Planning. In *20th Annual IFIP Networking Conference 2021 (IFIP Networking 2021)*. Helsinki, Finland.
- [23] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. 2018. Netcomplete: Practical Network-Wide Configuration Synthesis with Autocompletion. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation* (Renton, WA, USA) (NSDI'18). USENIX Association, USA, 579–594.
- [24] William Enck, Patrick McDaniel, Subhabrata Sen, Panagiotis Sebos, Sylke Sporel, Albert Greenberg, Sanjay Rao, and William Aiello. 2007. Configuration Management at Massive Scale: System Design and Experience. In *2007 USENIX Annual Technical Conference (USENIX ATC 07)*. USENIX Association, Santa Clara, CA. <https://www.usenix.org/conference/2007-usenix-annual-technical-conference/configuration-management-massive-scale-system>
- [25] Seyed K. Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. 2016. Efficient Network Reachability Analysis Using a Succinct Control Plane Representation. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) (OSDI'16). USENIX Association, USA.
- [26] Nick Feltovich. 2003. Nonparametric Tests of Differences in Medians: Comparison of the WilcoxonMann-Whitney and Robust Rank-Order Tests. *Experimental Economics* (2003).
- [27] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. 2015. A General Approach to Network Configuration Analysis. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Oakland, CA) (NSDI'15). USENIX Association, USA, 469–483.
- [28] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. 2016. Fast Control Plane Analysis Using an Abstract Representation. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) (SIGCOMM '16). Association for Computing Machinery, New York, NY, USA, 300–313. <https://doi.org/10.1145/2934872.2934876>
- [29] Milad Ghaznavi, Elaheh Jalalpour, Bernard Wong, Raouf Boutaba, and Ali José Mashtizadeh. 2020. Fault Tolerant Service Function Chaining. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (Virtual Event, USA) (SIGCOMM '20). Association for Computing Machinery, New York, NY, USA, 198–210. <https://doi.org/10.1145/3387514.3405863>
- [30] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. 2016. Evolve or Die: High-Availability Design Principles Drawn from Googles Network Infrastructure. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) (SIGCOMM '16). ACM, New York, NY, USA, 58–72. <https://doi.org/10.1145/2934872.2934891>
- [31] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. 2014. Dynamic Scheduling of Network Updates. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (Chicago, Illinois, USA) (SIGCOMM '14). ACM, New York, NY, USA, 539–550. <https://doi.org/10.1145/2619239.2626307>
- [32] Siva Kesava Reddy Kakarla, Ryan Beckett, Behnaz Arzani, Todd Millstein, and George Varghese. 2020. GRoot: Proactive Verification of DNS Configurations. In *SIGCOMM 2020*. <https://www.microsoft.com/en-us/research/publication/groot-proactive-verification-of-dns-configurations/> Best Paper Award.
- [33] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking for Networks. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (San Jose, CA) (NSDI'12). USENIX Association, USA.
- [34] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. 2010. Onix: A Distributed Control Platform for Large-Scale Production Networks. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada) (OSDI'10). USENIX Association, USA, 351–364.
- [35] John R. Lanzante. 1996. Resistant, Robust and Non-parametric Techniques for the analysis of climate data: Theory and Examples, including applications to Historical Radiosonde Station. *International Journal of Climatology* (1996).
- [36] Ze Li, Qian Cheng, Ken Hsieh, Yingnong Dang, Peng Huang, Pankaj Singh, Xinsheng Yang, Qingwei Lin, Youjiang Wu, Sebastian Levy, and Murali Chintalapati.

2020. Gandalf: An Intelligent, End-To-End Analytics Service for Safe Deployment in Large-Scale Cloud Infrastructure. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 389–402. <https://www.usenix.org/conference/nsdi20/presentation/li>
- [37] Hongqiang Harry Liu, Xin Wu, Wei Zhou, Weiguo Chen, Tao Wang, Hui Xu, Lei Zhou, Qing Ma, and Ming Zhang. 2018. Automatic Life Cycle Management of Network Configurations. In *Proceedings of the Afternoon Workshop on Self-Driving Networks (Budapest, Hungary) (SelfDN 2018)*. Association for Computing Machinery, New York, NY, USA, 29–35. <https://doi.org/10.1145/3229584.3229585>
- [38] Yujie Liu, Yong Li, Yue Wang, and Jian Yuan. 2015. Optimal Scheduling for Multi-flow Update in Software-Defined Networks. *Journal of Network and Computer Applications* 54, C (Aug. 2015), 11–19. <https://doi.org/10.1016/j.jnca.2015.04.009>
- [39] 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 *ACM CoNEXT*.
- [40] 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 *ACM CoNEXT*.
- [41] Ajay Mahimkar, Han Hee Song, Zihui Ge, Aman Shaikh, Jia Wang, Jennifer Yates, Yin Zhang, and Joanne Emmons. 2010. Detecting the Performance Impact of Upgrades in Large Operational Networks. In *ACM SIGCOMM*.
- [42] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. 2011. Debugging the Data Plane with Anteater. *SIGCOMM Comput. Commun. Rev.* 41, 4 (Aug. 2011), 290–301.
- [43] MiniZinc. 2020. MiniZinc - a free and open constraint modeling language. <https://www.minizinc.org>. Accessed on 2019-09-16.
- [44] Jeffrey C. Mogul, Alvin AuYoung, Sujata Banerjee, Lucian Popa, Jeongkeun Lee, Jayaram Mudigonda, Puneet Sharma, and Yoshio Turner. 2013. Corybantic: Towards the Modular Composition of SDN Control Programs. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks (College Park, Maryland) (HotNets-XII)*. ACM, New York, NY, USA, Article 1, 7 pages. <https://doi.org/10.1145/2535771.2535795>
- [45] NetConf. 2021. Network Configuration Protocol (NETCONF). Retrieved January 26, 2021 from <https://en.wikipedia.org/wiki/NETCONF>
- [46] Binh Nguyen, Zihui Ge, Jacobus Van der Merwe, He Yan, and Jennifer Yates. 2015. ABSENCE: Usage-Based Failure Detection in Mobile Networks. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking (Paris, France) (MobiCom '15)*. 464–476. <https://doi.org/10.1145/2789168.2790127>
- [47] Laurent Perron and Vincent Furnon. 2019. OR-Tools. Retrieved January 26, 2021 from <https://developers.google.com/optimization>
- [48] Santhosh Prabhu, Kuan-Yen Chou, Ali Kheradmand, P. Brighten Godfrey, and Matthew Caesar. 2020. Plankton: Scalable network configuration verification through model checking. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA.
- [49] Chaitan Prakash, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akella, Sujata Banerjee, Charles Clark, Yadi Ma, Puneet Sharma, and Ying Zhang. 2015. PGA: Using Graphs to Express and Automatically Reconcile Network Policies. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (London, United Kingdom) (SIGCOMM '15)*. Association for Computing Machinery, New York, NY, USA, 29–42. <https://doi.org/10.1145/2785956.2787506>
- [50] Mubashir Adnan Qureshi, Ajay Mahimkar, Lili Qiu, Zihui Ge, Max Zhang, and Ioannis Broustis. 2017. Coordinating rolling software upgrades for cellular networks. In *25th IEEE International Conference on Network Protocols, ICNP 2017, Toronto, ON, Canada, October 10-13, 2017*. 1–10. <https://doi.org/10.1109/ICNP.2017.8117537>
- [51] Shambwaditya Saha, Santhosh Prabhu, and P. Madhusudan. 2015. NetGen: Synthesizing Data-Plane Configurations for Network Policies. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (Santa Clara, California) (SOSR '15)*. Association for Computing Machinery, New York, NY, USA, Article 17, 6 pages. <https://doi.org/10.1145/2774993.2775006>
- [52] Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger. 2011. Diagnosing performance changes by comparing request flows. In *USENIX NSDI*.
- [53] S. Siegel and N. J. Jr. Castellan. 1998. Nonparametric Statistics for the Behavioral Sciences. *New York: McGraw-Hill* (1998).
- [54] Samuel Steffen, Timon Gehr, Petar Tsankov, Laurent Vanbever, and Martin Vechev. 2020. Probabilistic Verification of Network Configurations. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (Virtual Event, USA) (SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 750–764. <https://doi.org/10.1145/3387514.3405900>
- [55] Peng Sun, Ratul Mahajan, Jennifer Rexford, Lihua Yuan, Ming Zhang, and Ahsan Arefin. 2014. A Network-state Management Service. In *Proceedings of the 2014 ACM Conference on SIGCOMM (Chicago, Illinois, USA) (SIGCOMM '14)*. ACM, New York, NY, USA, 563–574. <https://doi.org/10.1145/2619239.2626298>

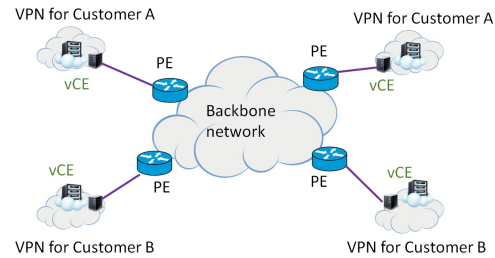


Figure 7: Virtual Private Network (VPN).

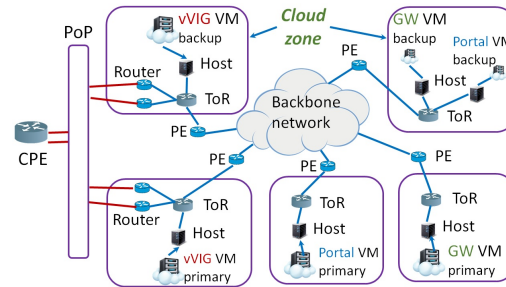


Figure 8: Software Defined Wide Area Network.

- [56] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky H.Y. Wong, and Hongyi Zeng. 2016. Robotron: Top-down Network Management at Facebook Scale. In *Proceedings of the 2016 ACM SIGCOMM Conference (Florianopolis, Brazil) (SIGCOMM '16)*. ACM, New York, NY, USA, 426–439. <https://doi.org/10.1145/2934872.2934874>
- [57] Aisha Syed, Bilal Anwer, Vijay Gopalakrishnan, and Jacobus Van der Merwe. 2019. DEPO: A Platform for Safe DEployment of Policy in a Software Defined Infrastructure (SOSR '19). 98–111. <https://doi.org/10.1145/3314148.3314358>
- [58] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatchalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. 2015. Holistic Configuration Management at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles (Monterey, California) (SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 328–343. <https://doi.org/10.1145/2815400.2815401>
- [59] Bingchuan Tian, Xinyi Zhang, Ennan Zhai, Hongqiang Harry Liu, Qiaobo Ye, Chunsheng Wang, Xin Wu, Zhiming Ji, Yihong Sang, Ming Zhang, Da Yu, Chen Tian, Haitao Zheng, and Ben Y. Zhao. 2019. Safely and Automatically Updating In-Network ACL Configurations with Intent Language. In *Proceedings of the ACM Special Interest Group on Data Communication (Beijing, China) (SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 214–226. <https://doi.org/10.1145/3341302.3342088>
- [60] Zengwen Yuan, Qianru Li, Yuanjie Li, Songwu Lu, Chunyi Peng, and George Varghese. 2018. Resolving Policy Conflicts in Multi-Carrier Cellular Access. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking (New Delhi, India) (MobiCom '18)*. ACM, New York, NY, USA, 147–162. <https://doi.org/10.1145/3241539.3241558>
- [61] Pamela Zave, Ronaldo A. Ferreira, Xuan Kelvin Zou, Masaharu Morimoto, and Jennifer Rexford. 2017. Dynamic Service Chaining with Dysco. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (Los Angeles, CA, USA) (SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 57–70. <https://doi.org/10.1145/3098822.3098827>

Appendices are supporting material that has not been peer-reviewed.

## A NETWORKS AND SERVICES

**Virtual private network (VPN).** In a VPN (Fig. 7), the customer network connects to the core backbone network via pairs of customer edge (CE) routers and provider edge (PE) routers. We typically have a mix of physical and virtualized PE and CE routers and they can share the same infrastructure, such as a core router.

**Software defined wide area networks (SDWAN).** In an SDWAN (Fig. 8), the customer typically connects using a customer premise

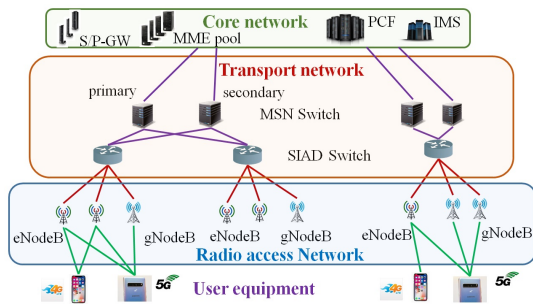


Figure 9: 4G and 5G service architecture.

equipment (CPE) with a service provider point of presence (PoP). The aggregate router connects the PoP with multiple cloud zones. Each cloud zone of the SDWAN has different network functions such as virtual gateway (vGW), portal, virtualized internet gateway (vVIG), top-of-rack (ToR) switches and physical servers hosting these virtualized network functions (vNFs). The virtual gateway handles the tunneling of traffic from the edge of the SDWAN portal. The vVIG is used to provide access to the vGWs hosted in the data center over an encrypted link. The portal is used for configuration and monitoring of the whole SDWAN ecosystem. The primary and backup vNFs can be in the same or different cloud zones.

**4G/5G cellular service.** Fig. 9 shows architecture for 4G/5G cellular. The end-users connect to the cellular radio access network via the base station (eNodeB in 4G and gNodeB in 5G). The base station consists of several radio channels on different carrier frequencies. The user equipment (UE) communicates with a specific radio channel on a base station. Depending on the signal strength and base station load, the UEs can automatically determine the channel to connect to. The base station connects via a switch with the transport and core networks. The cellular core consists of several network function types such as policy charging function (PCF), gateways (S/P-GW), mobility management entity (MME) and multimedia services (IMS).

## B DYNAMIC COMPOSITION EXAMPLES

Listing 1 depicts one full example of high-level user input. Such JSON code is submitted to CORNET REST API from the GUI (Graphic User Interface). We have pruned control content such as session information, callbacks, and other tracking mechanisms.

Listing 1: High-level optimization intent user API.

```

1 {
2   "scheduling_window": {
3     "start": "2020-07-01 00:00:00",
4     "end": "2020-07-07 23:59:00",
5     "granularity": {"metric": "day", "value": 1}
6   },
7   "maintenance_window": {
8     "start": "0:00",
9     "end": "6:00",
10    "granularity": "hour",
11    "timezone": "local"
12  },
13  "excluded_periods": [
14    {
15      "start": "2020-07-01 00:00:00",
16      "end": "2020-07-01 23:59:00"
17    },
18    {
19      "start": "2020-07-04 00:00:00",
20      "end": "2020-07-05 23:59:00"
21    }
22  ],
23  "schedulable_attribute": "common_id",
24  "conflict_attribute": "common_id",

```

```

25 "inventory": [...],
26 "frozen_elements": [
27   {
28     "common_id": "id00041"
29   },
30   {
31     "common_id": "id00283",
32     "start": "2020-01-13 00:00:00",
33     "end": "2020-01-13 00:00:00"
34   },
35   {
36     "market": "NYC",
37     "start": "2020-07-03 00:00:00",
38     "end": "2020-07-06 00:00:00"
39   },
40   ...
41 ],
42 "conflict_table": {
43   "id000001": [
44     {
45       "start": "2020-07-01 00:00:00",
46       "end": "2020-07-04 00:00:00",
47       "tickets": ["CHG000005482383"]
48     },
49     {
50       "start": "2019-07-07 00:00:00",
51       "end": "2019-07-15 00:00:00",
52       "tickets": ["CHG000005485234"]
53     }
54   ],
55   "id000002": [
56     {
57       "start": "2020-07-03 00:00:00",
58       "end": "2020-07-05 00:00:00",
59       "tickets": ["CHG000005485234", "CHG000005485999"]
60     }
61   ],
62   ...
63 },
64 "constraints": [
65   {
66     "name": "conflict_handling",
67     "value": "minimize-conflicts"
68   },
69   {
70     "name": "concurrency",
71     "base_attribute": "common_id",
72     "operator": "<=",
73     "granularity": {"metric": "day", "value": 1},
74     "default_capacity": 300
75   },
76   {
77     "name": "concurrency",
78     "base_attribute": "market",
79     "operator": "<=",
80     "granularity": {"metric": "day", "value": 1},
81     "default_capacity": 5
82   },
83   {
84     "name": "concurrency",
85     "base_attribute": "common_id",
86     "aggregate_attribute": "pool_id",
87     "operator": "<=",
88     "granularity": {"metric": "day", "value": 1},
89     "default_capacity": 10
90   },
91   {
92     "name": "uniformity",
93     "attribute": "timezone",
94     "value": 1
95   },
96   {
97     "name": "localize",
98     "attribute": "market"
99   }
100 ]
101 }

```

In lines 2–12, the user defines the period when they want to schedule the changes and define the maintenance window interval, which determines, in turn, the schedulable timeslot. Note that in this example, the timeslot is defined as a single night since the scheduling should be performed daily from 0:00 to 6:00 local time. Lastly, multiple constraints may specify different time granularities. For example, suppose that the timeslot is defined as one day. We may have a concurrency constraint dictating a bound in the number of elements scheduled in one day and another concurrency constraint defining a limit on the number of elements per week.

Within the scheduling window, we may have some excluded periods described on lines 13–22. Such intervals are useful to define holidays and special events such as the Super Bowl or the presidential inauguration, where the network should not be touched.

The user then defines the elementary schedulable attribute (ESA) and the conflict attribute (CA) (lines 23–24). The ESA can define unique items or groups of items, so that CORNET must take its multiplicity in the account, depending on the user’s desires. For instance, the concurrency constraint may be specified in terms of the actual number of elements or another attribute, e.g., the number of markets that can be scheduled concurrently. There is also the hybrid situation where a user may want to schedule markets but specifies concurrency in terms of the number of elements, in which case we assign a weight to each market equal to its number of elements. Similarly, the user defines the CA, which, in general, is the same as the ESA. However, in hybrid situations, CORNET must account for the relationship between the ESA and the CA, in creating mappings. Using the previous example, if the ESA is “market” and the CA is “eNodeB,” when scheduling a market, CORNET must look for all conflicts generated by all eNodeBs within a market.

Line 25 names the inventory, which can be a query to an inventory database (not shown for brevity). Following, lines 26–41 define elements (and periods) that cannot be touched or scheduled. Note that both ESA and non-ESA attributes can be listed. In case of non-ESA, CORNET must do the proper mapping to the ESA. Note that frozen\_element is similar to excluded\_periods, but provides finer control because it can be set for each element (or group of elements through non-ESA). While excluded\_periods avoid any element to be scheduled (and therefore, such period must be ignored by the optimization, which, generally, makes the problem easier to solve), frozen\_element may introduce cross-dependencies among frozen and not-frozen elements, and the optimizer must take such relationships in the account during the optimization.

Lines 42–63 shows the conflict table, usually extracted automatically from a ticketing system based on the conflict scope. Each element may have multiple conflicts in the same or different periods. Lines 64–102 show the user-defined constraints. In lines 65–68, the user chooses whether they want a conflict-free schedule (and risk having unscheduled nodes or a longer makespan), or if they want to schedule as many nodes as possible but minimize the number of generated conflicts (this is useful for high-priority changes).

Next, we have three concurrency constraints, one uniformity, and one localize constraint. The first concurrency constraint (lines 69–75) defines a global limit on the number of scheduled common\_ids per timeslot/maintenance window. Since the schedule can spread over a large region, the user also limits the number of markets touched per timeslot (lines 76–82). In lines 83–90, we assume that a common\_id belongs to a pool, and the user limits the number of scheduled common\_ids per pool in a given night. The user also requires that CORNET does not schedule nodes that are two or more timezones apart in the same night. However, neighboring timezones are allowed (lines 91–95). Finally, the user also requests the non-interleaving of nodes from different markets, i.e., when scheduling common\_ids from a market, finish that market before going to the next (lines 96–99).

Listing 2 depicts the mathematical representation using Minizinc. We omit all data for the sake of brevity. Note that CORNET must build several ESA to non-ESA mappings, both from parameter data and decision variables. Indeed, CORNET also handles non-ESA to non-ESA mappings, carefully taking into account multiplicity and duplicates. Some structures may look unnecessary

from the viewpoint of an operations research/optimization practitioner. For example, both decision variables MARKET\_SCHEDULED and POOL\_ID\_SCHEDULED are defined as a function of COMMON\_ID\_SCHEDULED, and therefore, we could rewrite the constraints depending on these variables only in terms of COMMON\_ID\_SCHEDULED. However, this approach makes the model very dense and hard to debug.

**Listing 2: Minizinc optimization model.**

```

1 %*****
2 % Data
3 %*****
4
5 set of int: bool_t = 0..1;
6 int: False = 0;
7 int: True = 1;
8
9 int: n_common_id = ...;
10 int: n_market = ...;
11 int: n_pool_id = ...;
12 int: n_timeslots = ...;
13 float: max_distance_ctr1 = 1;
14
15 % common_id -> pool_id
16 int: n_common_id2pool_id_matrix = 5;
17 array[1..n_common_id2pool_id_matrix, 1..2] of int:
18   common_id2pool_id_matrix = ...;
19
20 % common_id -> market_id
21 int: n_common_id2market_matrix = 5;
22 array[1..n_common_id2market_matrix, 1..2] of int:
23   common_id2market_matrix = ...;
24
25 array[1..n_common_id] of int: common_id_weight = ...;
26 array[1..n_market] of int: market_weight = ...;
27 array[1..n_common_id] of float: common_id_timezone_data = ...;
28
29 int: n_common_id_timeslots_conflicts = ...;
30 array[1..n_common_id_timeslots_conflicts, 1..3] of int:
31   common_id_conflict_table = ...;
32
33 int: n_market_timeslots_conflicts = 5;
34 array[1..n_market_timeslots_conflicts, 1..3] of int:
35   market_conflict_table = ...;
36
37 int: n_common_id_frozen_timeslots = ...;
38 array[1..n_common_id_frozen_timeslots, 1..2] of int:
39   common_id_frozen_timeslots = ...;
40
41 int: n_timeslots_ctr1 = n_timeslots;
42 array[1..n_timeslots_ctr1] of int:
43   common_id_capacity_ctr1 = ...;
44
45 int: n_timeslots_ctr2 = n_timeslots;
46 array[1..n_timeslots_ctr2] of int:
47   market_capacity_ctr2 = ...;
48
49 int: n_timeslots_ctr3 = n_timeslots;
50 array[1..n_timeslots_ctr3] of int:
51   common_id_pool_id_capacity_within_ctr3 = ...;
52
53 %*****
54 % Decision variables
55 %*****
56
57 array[1..n_common_id, 1..n_timeslots] of var bool_t:
58   COMMON_ID_SCHEDULED :: add_to_output;
59
60 array[1..n_market, 1..n_timeslots] of var bool_t:
61   MARKET_SCHEDULED :: add_to_output =
62   array2d(1..n_market, 1..n_timeslots,
63     [bool2int(
64       sum(i in 1..n_common_id2market_matrix
65         where common_id2market_matrix[i,2] == j)
66         (COMMON_ID_SCHEDULED[common_id2pool_id_matrix[i,1],t])
67         >= 1
68       ) | j in 1..n_pool_id, t in 1..n_timeslots
69     ]
70   );
71
72 var int: COMMON_ID_NUM_CONFLICTS :: add_to_output =
73   sum(i in 1..n_common_id_timeslots_conflicts) (
74     COMMON_ID_SCHEDULED[common_id_conflict_table[i,1],
75       common_id_conflict_table[i,2]]
76     * common_id_conflict_table[i,3]
77   );
78
79 array[1..n_common_id] of var 0..n_timeslots:
80   COMMON_ID_ASSIGNED_TIMESLOT :: add_to_output =
81   [sum(j in 1..n_timeslots) (j * COMMON_ID_SCHEDULED[i,j])
82     | i in 1..n_common_id];
83
84 array[1..n_market] of var opt 0..n_timeslots:
85   MARKET_START_TIME :: add_to_output =
86   array1d(1..n_market,
87     [min([t | t in 1..n_timeslots where
88       sum(k in 1..n_common_id2market_matrix
89         where common_id2market_matrix[k,2] == j)

```

```

90         (COMMON_ID_SCHEDULED[common_id2market_matrix[k,1],t])
91         >= 1
92     ]) | j in 1..n_market,
93 ];
94 );
95
96 array[1..n_market] of var opt 0..n_timeslots:
97 MARKET_END_TIME :: add_to_output =
98 array1d(1..n_market,
99 [max([t | t in 1..n_timeslots where
100     sum(k in 1..n_common_id2market_matrix
101     where common_id2market_matrix[k,2] == j)
102     (COMMON_ID_SCHEDULED[common_id2market_matrix[k,1],t])
103     >= 1
104     ]) | j in 1..n_market,
105 ];
106 );
107
108 *****
109 % Primary constraints
110 *****
111
112 constraint
113 forall(i in 1..n_common_id) (
114     sum(j in 1..n_timeslots)(COMMON_ID_SCHEDULED[i,j]) <= 1
115 );
116
117 constraint
118 forall(i in 1..n_common_id_timeslots_conflicts) (
119     COMMON_ID_SCHEDULED[common_id_conflict_table[i,1],
120     common_id_conflict_table[i,2]]
121     <= common_id_conflict_table[i,3] /
122     (common_id_conflict_table[i,3] + 1)
123 );
124
125 constraint
126 forall(i in 1..n_common_id_frozen_timeslots) (
127     COMMON_ID_SCHEDULED[
128     common_id_frozen_timeslots[i,1],
129     common_id_frozen_timeslots[i,2],
130     ] == 0;
131 );
132
133 *****
134 % User constraints
135 *****
136
137 constraint
138 forall(j in 1..n_timeslots_ctr01) (
139     sum(i in 1..n_common_id) (
140     common_id_weight[i] * COMMON_ID_SCHEDULED[i,j]
141     ) <= common_id_capacity_ctr01[j]
142 );
143
144 constraint
145 forall(j in 1..n_timeslots_ctr02) (
146     sum(i in 1..n_market) (
147     market_weight[i] * MARKET_SCHEDULED[i,j]
148     ) <= market_capacity_ctr02[j]
149 );
150
151 constraint
152 forall(j in 1..n_timeslots, k in 1..n_pool_id) (
153     sum(i in 1..n_common_id2pool_id_matrix
154     where common_id2pool_id_matrix[i,2] == k)
155     (COMMON_ID_SCHEDULED[common_id2pool_id_matrix[i,1], j])
156     <= common_id_pool_id_capacity_within_ctr03[j]
157 );
158
159 constraint
160 forall(t in 1..n_timeslots) (
161     forall(i in 1..n_common_id, j in 1..n_common_id where i < j) (
162     abs(common_id_timezone_data[i] -
163     common_id_timezone_data[j]) *
164     (COMMON_ID_SCHEDULED[i,t] * COMMON_ID_SCHEDULED[j,t])
165     <= max_distance_ctr1 - 1
166     );
167 );
168
169 constraint
170 forall(i,j in 1..n_market where i < j) (
171     absent(MARKET_START_TIME[i]) \/\
172     absent(MARKET_START_TIME[j]) \/\
173     absent(MARKET_END_TIME[i]) \/\
174     absent(MARKET_END_TIME[j]) \/\
175     deopt(MARKET_START_TIME[i]) >= deopt(MARKET_END_TIME[j]) \/\
176     deopt(MARKET_END_TIME[i]) <= deopt(MARKET_START_TIME[j])
177 );
178
179 *****
180 % Objective function
181 *****
182
183 float: BIGM = max(common_id_capacity_ctr01) *
184     ((n_timeslots * (n_timeslots + 1)) div 2);
185
186 solve minimize
187 BIGM * COMMON_ID_NUM_CONFLICTS +
188 - sum[
189     (n_timeslots - j + 1.0) *
190     sum([COMMON_ID_SCHEDULED[i,j] | i in 1..n_common_id]
191     | j in 1..n_timeslots
192     )];

```

## C CUSTOM HEURISTIC FOR SCALABLE OPTIMIZATION

eNodeB/gNodeB operations teams incorporate a wide variety of constraints for their change schedule discovery ranging from (a) concurrency constraints on the EMS (Element Management System) for capturing how many concurrent executions of changes are permissible via the EMS, (b) distinct assignment of nodes across TAC<sup>2</sup> (Tracking Area Codes), (c) consistency constraint on the USID<sup>3</sup> for ensuring scheduling of both eNodeB and gNodeB within the same USID on the same time-slot for software compatibility, (d) uniformity constraint on time-zone and hardware version for ensuring safe execution of the changes within the chosen time-slots, (e) localize constraint on market to simplify performance impact assessment post deployments, and (f) conflict constraint on the same eNodeB/gNodeB as well as the connected SIAD.

We use the following notations: (i) Set  $\mathcal{L}$  of USIDs/Locations; (ii) Set  $\mathcal{T}$  of Tracking Area codes (TAC); (iii) Set  $\mathcal{M}$  of markets; (iv) Set  $\mathcal{Z}$  of timezones or UTC Offsets; (v) Set  $\mathcal{E}$  of Element Management Systems (EMS); (vi) Maximum schedule size  $max\_timeslots \in \mathbb{N}^+$ ; and (vii) Capacity  $C(s) \geq 0$  for each timeslot  $s = 1, \dots, max\_timeslots$ .

We assume the generic mapping  $agg\_attrib[base\_attrib]$  to list all elements of  $agg\_attrib$  for each  $base\_attrib$ . For instance, consider a market  $m \in \mathcal{M}$ . So,  $tacs[m]$  lists all TACs associated to market  $m$ . Let  $cap(s) \geq 0$  be the remaining capacity for timeslot  $s$ . Let  $num\_confl(\mathcal{S}) \geq 0$  be the number of conflicts for schedule  $\mathcal{S}$ . The weighted total completion time for schedule  $\mathcal{S}$  is defined as

$$wtct(\mathcal{S}) = \sum_{s \in timeslots(\mathcal{S})} s \times num\_sched(s, \mathcal{S}) \quad (6)$$

where  $timeslots(\mathcal{S})$  lists all the timeslots used on the schedule  $\mathcal{S}$  and  $num\_sched(s, \mathcal{S})$  gives the number of nodes scheduled on timeslot  $s$ .

Therefore, we sort the timezones by the UTC offset (e.g., Eastern zone would have UTC offset of -4 or -5 based on daylight savings) and schedule each timezone independently and sequentially. Note that we allow nodes from adjacent time zones to be scheduled in the "border" timezones, i.e., at the beginning or end of each timezone schedule with spare capacity. In theory, we could interleave nodes from adjacent timezones for a tighter schedule. However, scheduling nodes with clear separation of timezones makes the schedule easier to deploy.

Once the main problem is decomposed, we apply Algorithm 1 for each sub-instance. In line 1, we start with an empty schedule so that its number of conflicts and the total weighted completion time are infinity. In the main loop (lines 2–23), until a stopping criterion is reached (maximum wall-clock time), the algorithm generates a permutation of markets (line 3), and sets the timeslot for the beginning of the schedule. We choose the market attribute for permutation because of the localize constraint on the market. Note that for the first sub-instance,  $start\_timeslot = 1$ . In the next optimization for the following sub-instance,  $start\_timeslot$  should be the last timeslot with spare capacity where we have a node scheduled in the current sub-schedule. Note that, we do some bookkeeping to track

<sup>2</sup>A TAC (Tracking Area Code) consists of multiple USIDs and a market consists of multiple TACs.

<sup>3</sup>A USID consists of cellular towers (gNodeB, eNodeB, NodeB) supporting multiple technologies (5G, LTE, UMTS, respectively).

**Algorithm 1:** Local search for eNodeBs/gNodeBs scheduling

---

```

1 Let  $S^*$  be the best solution so far so that  $wtct(S^*) \leftarrow \infty$  and
   $num\_confl(S^*) \leftarrow \infty$ ;
2 while stopping criteria is not reached do
3   Let  $M$  be a market permutation of  $\mathcal{M}$ ;
4    $curr\_ts \leftarrow start\_timeslot$ ;
5   foreach  $m \in M$  in the given sequence do
6      $rem\_tacs \leftarrow tacs[m]$ ;
7     while  $rem\_tacs$  is not empty do
8       if  $curr\_ts = max\_timeslots$  then
9         Push the remaining nodes from  $rem\_tacs$  as leftovers;
10        Go to line 21;
11      Sort  $rem\_tacs$  on a non-decreasing order number of conflicts
        on timeslot  $curr\_ts$ , then by non-increasing order of number
        of nodes;
12      foreach  $t \in rem\_tacs$  in the given sequence do
13        For each  $\ell \in usids[t]$ , schedule all nodes from  $\ell$ 
        timeslot  $curr\_ts$  respecting and adjusting the
        capacity  $cap[curr\_ts]$ ;
14        if all nodes of  $t$  are scheduled then
15           $rem\_tacs \leftarrow rem\_tacs \setminus \{t\}$ ;
16        if  $cap[curr\_ts] = 0$  then
17           $curr\_ts \leftarrow curr\_ts + 1$ ;
18          Got to line 7;
19        if  $curr\_ts = max\_timeslots$  then
20          Go to line 9;
21   Let  $S$  be the generated schedule;
22   if  $\langle num\_confl(S), wtct(S) \rangle < \langle num\_confl(S^*), wtct(S^*) \rangle$  then
23      $S^* \leftarrow S$ ;
24 return the best schedule  $S^*$ .

```

---

spare capacities and non-adjacent timezones, which we omit here for brevity.

Once the market permutation is set, our algorithm iterates over each market (lines 5–20), trying to schedule the TACs for that market. A market typically consists of thousands of eNodeBs/gNodeBs. The intuition behind selecting the TAC attribute within a market is that a TAC consists of eNodeBs/gNodeBs (on the order of hundreds) geographically close to each other. By bringing down the problem at the TAC level, we are able to handle the schedule discovery for hundreds of nodes within a TAC and then scaling this up-to the market, EMS and thus the whole network comprising of hundreds of thousands of eNodeBs and gNodeBs. In lines 8–10, the algorithm checks whether we have reached the end of the scheduling window and if so, all remaining nodes are put aside as leftovers. Next, we sort the TACs by the number of conflicts on the current timeslot  $curr\_ts$  and, then, their size (line 11). The intuition is that we want to schedule less-conflicting large TACs as soon as possible. Using this order, the algorithm schedules nodes in the same USID from this TAC so that nodes belonging to the same USID are always scheduled in the same timeslot (line 13). Again, for the sake of brevity, we omit several details, such as how to update the timeslot capacities. When the timeslot capacity is reached, the algorithm starts scheduling nodes the next timeslot (lines 16–18). If we reach the scheduling window’s end, we again put aside the leftover nodes (lines 19–20). After all the markets are scheduled, we compare the total number of conflicts (first) and the weighted total completion time (second) with the incumbent best solution

and update accordingly (lines 22–23). For the leftover nodes, one would have to submit a new request in a different scheduling time window.

**D COMPOSITION EVALUATION FOR CHANGE IMPACT VERIFICATION****Table 5:** CORNET flexible composition evaluation for change impact verification.

KPI group	KPIs	Tables	No join	2-way join	3-way join
Scorecard	9	6	6	0	0
Level-1	58	17	14	3	0
Level-2	123	14	10	3	1
Level-3	159	17	16	1	0
All (of above)	349	48	40	7	1

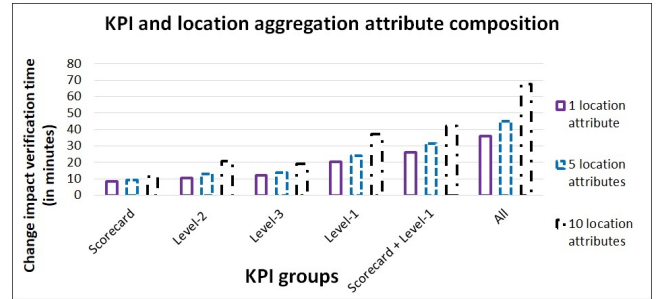
**Figure 10:** Impact verification time as a function of KPI composition and location aggregation attribute composition. The number of nodes is 400.

Fig.10 shows the impact verification time as a function of KPI composition (using Table 5) and location aggregation attribute composition. The location aggregation attributes are constructed using the eNodeB inventory and configuration. The number of location attributes are varied between 1, 5 and 10. We fix the number of nodes (or, eNodeBs) to be 400. As can be seen from Fig.10, the impact verification time increases expectedly as the number of location attributes increases. Also, as the KPI compositions and group increase, the number of tables (no-join, 2-way / 3-way joins) increases and thus the verification time increases accordingly.

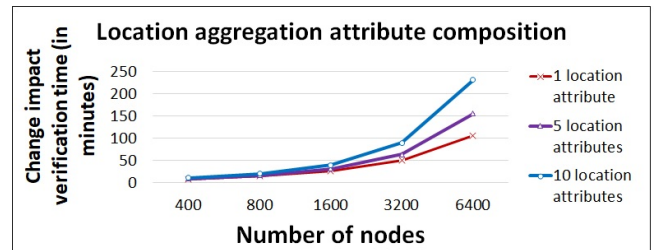
**Figure 11:** Impact verification time as a function of location aggregation attribute composition and number of nodes.

Fig. 11 shows that the impact verification time increases expectedly as the number of eNodeBs increases from 400 to 6400. This is influenced by the number of threads we create to deal with the number of nodes.

## E ADDITIONAL OPERATIONAL EXPERIENCES

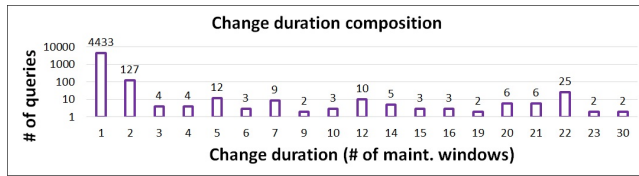


Figure 12: Change duration (in terms of maintenance windows) input across different scheduling request queries.

### E.1 Composition in change schedule planning

Figure 12 captures the flexible selection of different change durations in terms of maintenance windows for each node. A large number of change scheduling requests (4433) have a change duration of one maintenance window. However, there were a small number of requests that spanned multiple maintenance windows. Node re-tuning, construction work would expectedly fall in this category because of larger time needed to complete their change activity that involves physically going out to the location and conducting the work. We also learned that the operations teams would reserve a large window for FFA even if only one maintenance window was sufficient to complete the change activity at a node because they would take extreme caution in ensuring that no other conflicting activities happen during that time for a cleaner pre/post impact verification. We re-used the schedule planning workflow as well as the individual building blocks to support different compositions. Thus, CORNET enabled a higher degree of code re-use.

### E.2 Efficient change schedule planning

Traditionally (before CORNET), the network operations teams would reserve long duration maintenance windows to complete changes such as hardware repair or construction work (e.g., new tower adds). With CORNET, they improved their process and started creating short duration maintenance windows for individual nodes. This enabled other work groups to effectively co-ordinate their change schedule plans, and thus resulting in an overall improvement of change roll-out across multiple groups. Table 6 shows that there is a significant reduction in the average and standard deviation of the duration for construction-related changes due to CORNET. However, software upgrades, configuration changes, and node re-tuning did not have a change in behavior because the responsible teams would always reserve the minimum time needed to complete their change execution.

Table 6: Duration of change activities measured in number of maintenance windows.

	Average with CORNET	Standard deviation with	Average without CORNET	Standard deviation without
Software upgrade	1.92	3.63	1.97	3.98
Configuration change	1.29	2.25	1.58	2.71
Node re-tuning	3.17	6.02	4.03	7.04
Construction work	3.78	19.09	4.06	36.91

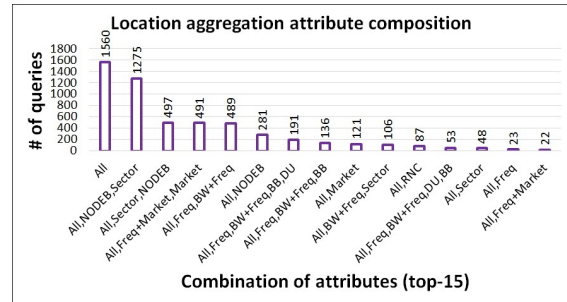


Figure 13: Number of location aggregation attributes selected by different users across different impact queries capturing the dynamic composition of attributes.

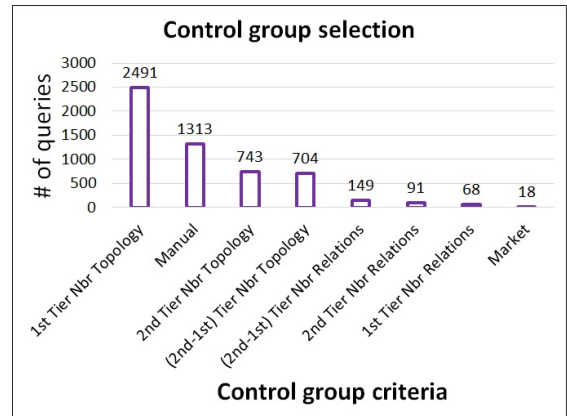


Figure 14: Different control groups selected by different users across different impact queries capturing the dynamic composition of control group selection criteria.

### E.3 Composition in change impact verification

We observed the operations teams usage patterns across different impact verifications requests and quantified the different compositions of the verification rules. Fig. 13 shows the different combination of location aggregation attributes used across different impact verification queries. Most of the queries would look at the time-aligned aggregate impact (All) for staggered roll-out and also the distribution of the impact across each (e/g)NodeB and sector (a sector is a part of base station and covers users and traffic on specific radio channels and carrier frequencies). Configuration attributes like carrier frequency (Freq), hardware versions (BB/DU) and inventory information like market were amongst the top combinations. Fig. 14 shows the different impact verification compositions across different control group definitions. Different changes can have different impact propagations (e.g., on the same node, 1-hop neighbor, 2-hop neighbors). We define 1<sup>st</sup> tier neighbors as a list of all the 1-hop away neighbors, 2<sup>nd</sup> tier neighbors as a list of all the 2-hop away neighbors and 2<sup>nd</sup> minus 1<sup>st</sup> is the difference between the two lists. The neighbors can be defined either using topological hierarchy or neighbor relations (e.g., X2 links in 4G for the eNodeBs). This highlights the importance and need of composition for the impact verification of network changes. Using CORNET, these compositions could be supported using the same impact verification workflow as well as individual building blocks demonstrating the benefit of code re-use.