

StepNet: A Compositional Framework with Reduced Querying for Homing Complex Network Services

Azzam Alsudais*, Shankaranarayanan Puzhavakath Narayanan[§], Bharath Balasubramanian[§], Zhe Huang[§], Eric Keller*
 *University of Colorado at Boulder, [§] AT&T Labs Research

Abstract—Homing or placement of network elements on cloud infrastructure is a crucial step in the orchestration of network services, involving complex interactions with several cloud and network service controllers. Network Service Providers (NSPs) currently follow a traditional approach akin to existing VM and VNF placement techniques that involves hand-crafting service specific heuristics for homing network services. However, operational experience from a Tier-1 NSP shows that existing approaches do not scale well when network services evolve and their requirements change. Further, these approaches require extensive and repetitive querying of the various controllers (e.g., to check customer eligibility or capacity), placing significant burden on the resources at the controllers. We propose *StepNet*, a compositional homing framework that allows service designers to easily mix and match homing requirements to create instances of the homing problem, enabling greater agility of service creation and evolution. *StepNet* adopts an incremental approach to querying that provides near optimal homing solutions, while reducing the cumulative time spent by all of the data sources responding to queries for each homing request (query cost). Our evaluation with production traces from a Tier-1 NSP shows a reduction in query cost of 92% for over 50% of the requests.

I. INTRODUCTION

Network Service Providers (NSPs) offer fundamental networking capabilities such as managed dedicated internet connectivity, Wide Area Networks, Virtual Private Networks, Voice Over IP, and Secure Cloud Connectivity. A critical step in provisioning these services for their customers is identifying either optimal locations for creating the network elements of the network service or reusing existing service instances (among several thousands providing the required capabilities) that can be shared among services. This process, referred to as the *Homing problem*¹, is performed based on a wide variety of constraints influenced both by the customer Service Level Objectives (SLOs) [1] and the NSP such as network latency, bandwidth capacity, and infrastructure capabilities.

Figure 1 shows an overview of the topology and homing requirements of a virtual Customer Premise Equipment (vCPE) residential broadband service, a simple but illustrative real-world network service offered by many NSPs. vCPE connects a residence to the vG at the Service Provider Edge (PE). The Bridged Residential Gateway (BRG) is the vCPE located at the residential customer premises, while the vGMux is a shared network service at the PE that maps layer-2 traffic between a subscriber's BRG and its unique vG, ensuring traffic isolation between multiple customers. In the homing

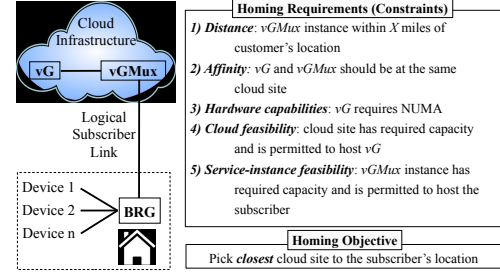


Fig. 1: Homing a Residential Broadband service.

terminology, the vCPE comprises of two *demands* - vG and vGMux, and two types of *candidates* - (i) PE cloud sites (called 'cloud' candidates), where new vG instances can be created, and (ii) existing instances of the vGMux service (called 'service' candidates), which can be shared by the new vCPE instance with other subscribers. The goal for homing the vCPE service, is to drive the selection of a close PE site to host the vG for a given customer, where an existing shared vGMux provides the required cross-connect capability. Finally, the homing requirements for the vCPE service are defined through five different constraints as shown in the figure.

Traditionally, NSPs have viewed homing as a constraint-based mapping of *resources* to *requirements* that has been explored in works on virtual machine (VM) placement [2]–[7], virtual network function (VNF) placement [8]–[11], capacity planning [12], facility location [13], [14] and replica placement in datastores [15], [16]. However, our detailed analysis of the service requirements and production homing traces of a Tier-1 NSP, reveal several challenges with this traditional approach as the number of network services and their operational complexity increases. In the next few paragraphs, we describe the two most significant challenges among them.

Evolving Service Requirements. Network services are complex with widely varying requirements that change as the service evolves. For example, the vCPE residential broadband service is typically offered with many pricing tiers, SLAs, and add-on features (e.g., added security) that alters the service requirements quite significantly. Developing hand-crafted heuristic optimization models for each service (and its variants) is a time-consuming and complex offline process with several cycles of testing and validation (order of months). Further, even simple updates to an existing service may often result in an entirely new formulation (e.g. linear to non-linear constraints) that will necessitate a significant amount of time before deployment. Clearly, this runs counter to the goal of evolving and maintaining systems in an agile manner.

¹Henceforth the word *Homing* refers to the Homing of network services
 978-3-903176-32-4 © 2021 IFIP

Aggregating Data during deployment. The homing service requires extensive interactions with tens to hundreds of SDN, Cloud, and Network Service controllers in order to identify a feasible placement decision. This results in repeated queries to the controllers to check for run-time factors such as customer eligibility to use certain service instances or the availability of capacity in a certain cloud-site. Since these controllers are primarily responsible for running and managing the life-cycle of these network services, NSPs rate-limit these repetitive homing related queries to the controllers, which limits the scale of the homing system as a whole. Our analysis of homing traces of a Tier-1 NSP showed that, if left unchecked, these queries would often exceed the allowed rate at the controllers (2x for $\sim 50\%$ of the time) and further, these controllers cumulatively spent more than 400 seconds per homing request to answer queries for $\sim 50\%$ of the homing requests. While pre-provisioning resources may help limit the need for such querying, in practice, resulting in massive over-provisioning and requires repetition when services evolve.

We address these challenges through a novel system, *StepNet* that is based on two key innovations. First, we introduce a novel compositional framework where homing instances of a service can be described through a declarative template that consists of *compositional blocks* through which a service designer can specify service requirements, including constraints and homing objectives. These abstractions have clearly defined structures, functional behavior, and APIs. These compositional blocks can be mixed and matched by service designers to create new homing requests with considerable ease as their customer requirements evolve.

Second, we introduce an online incremental approach to the homing problem which is designed to minimize the number of queries made to the controllers, while maintaining good solution quality. It does this through two key approaches: (i) rather than evaluating the entire solution space, we start with a small set of potential solutions, ordered based on the objective value, and gradually expand this set until a good solution is found, (ii) we sequentially evaluate the feasibility of constraints, performing the least expensive (in terms of queries) first, such that we can prune candidates before needing to evaluate the more expensive constraints.

We evaluate *StepNet* guided by production traces and 12 network services obtained from a Tier-1 NSP. Using the compositional framework, we generated over 1200 variants of these services supporting varied constraints and demands with just a few days of work. We demonstrate agile service evolution by supporting new run-time objectives and common heuristics for optimization with a few hundred lines of code. Next, we show that the incremental approach adopted by *StepNet*, reduces the cumulative time spent by all of the data sources responding to queries for each homing request (*i.e.*, query cost) by $> 1,000$ seconds for 50% of homing requests, and by $> 10,000$ seconds for 20% of homing requests, while maintaining close to optimal solution quality.

In summary, this paper makes the following contributions:

- Highlighting critical challenges in homing gleaned from

our detailed analysis of the service requirements and the production homing traces of a Tier-1 NSP (§II).

- A novel system, called *StepNet*, that addresses these challenges through a compositional framework (§III) for designing homing instances and an incremental approach to solving them (§IV).
- Extensive evaluation using production traces, proving *StepNet*'s efficacy (§V).

II. BACKGROUND AND MOTIVATION

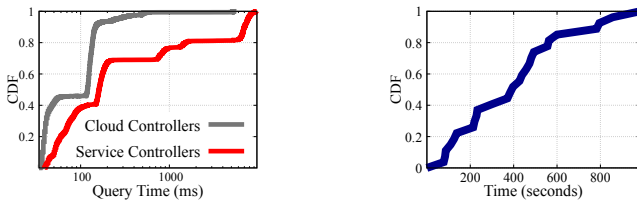
Typically, the process of homing in NSPs involves two distinct phases - (1) an offline “design” phase when the network service-specific homing heuristics are built and an estimated set of resources are pre-provisioned for homing an anticipated number of service instances, and (2) a “run-time” phase where the actual homing decisions are made for each service instance as they are created by the service orchestrators. We now describe the current approaches adopted by NSPs for homing, highlighting the challenges observed with these approaches based on interactions and operational experience with a Tier-1 NSP.

A. Design Phase: Challenge of Evolving Services

In the offline or design phase of the homing process the service developer generates the service model, *i.e.*, service demands, constraints and objectives described in §I, and develops service-specific optimization models and heuristics for homing the service, drawing from several works on VM and VNF placement [3]–[11], [17], [18]. These works typically formulate the placement problem using integer linear programming (ILP) or mixed integer linear programming (MILP), and propose tailor-made heuristics to relax and solve the problem for a specific use-case (*e.g.*, 5G network slicing).

Our discussions with a Tier-1 NSP revealed that this approach of developing service-specific heuristics, validating and testing them before production deployment, requires a significant Time To Market (TTM) (order of months) and resources. Further, services are often updated, creating new variants of them. For example, consider the case when a NSP wants to provide a new price-tiered offering or add a new firewall function for some subscribers. This requires additional constraints or demands that affect both the heuristic optimization models (*e.g.*, from a linear filtering constraint to a quadratic coupling constraint) and the underlying formulation of the homing heuristic. The new heuristic has to go through the entire cycle of testing and validation before production deployment, requiring a considerable amount of time. Clearly, the current approach compromises both evolvability and maintainability of network services, negating the benefits of virtualization.

Work in the peripheral space of optimizing software-defined networks seeks to simplify the optimization formulation process. For instance, SOL [19] and Chopin [20] provide a limited set of high-level APIs (*e.g.*, add link capacity constraint) to software-defined networking (SDN) applications to efficiently manage network resources. SDN applications, then, need to consume those APIs, and the framework will model those



(a) Individual query latency by type of controller.

(b) Total Time spent by controllers per homing request.

Fig. 2: Homing queries analysis for a week-long trace of a Tier-1 NSP.

high-level API calls as LP/ILP programs, and then solve them to find the best solution. However, these works assume complete knowledge when performing the optimization, so the addition or change of one service would require re-doing the whole optimization. VNF placement approaches [8]–[11], [17], [18] have the same disadvantage.

As evidenced above, the evolutionary nature of network services makes it impractical to design a specific model and heuristic for each possible service (and its variants). To address this problem, we present a compositional framework for homing in §III where new *demands* and *constraints* can be added with minimal development effort, different heuristics can be utilized with no changes to the service model, and service models can be added or changed without affecting existing services.

B. Run-time phase: Challenge of Aggregating Data

The online phase of homing begins when a new network service instance needs to be created by a service orchestrator, which invokes the homing service along with instance specific run-time inputs like Subscriber ID, Subscriber Location, *etc.* The homing service retrieves the pre-built models and heuristics from a model repository, and aggregates the data required by the models to run the heuristic on a solver like CPLEX [21]. The homing recommendations are then returned to the orchestration workflow that creates and configures these network elements based on these recommendations.

As we can see, a key aspect of the online phase is aggregating the data required for solving the homing heuristics. One approach used in practice by NSPs involves estimating the resources required for deploying an anticipated number of service instances and then reserving capacities for the tenant in the cloud. These pre-provisioned resources are inventoried in a centralized location, ready to be allocated through their respective Cloud and Service controllers when new service instances are created. This approach, however, suffers from two major problems: (i) reserving resources based on expected load across instances of the service often results in tremendous over-allocation, and (ii) as services evolve, this step needs to be performed repeatedly.

An alternative involves collecting the data on demand, which requires querying various *data-sources*, which include different inventories (for relatively static information), and multiple cloud and service controllers (for dynamic information). For example, consider the constraints for the vCPE service (Figure 1). Relatively static information such as *Distance*,

Affinity and *Hardware capability* can be maintained in inventories/databases. However, the *cloud feasibility* constraint, evaluated at the cloud controller is inherently dynamic, and requires run-time parameters like subscriber information (*e.g.*, customer-key) and service instance information (*e.g.*, tenant-id) to evaluate whether a given cloud instance can support creating a new network element of the service for the subscriber. Similarly, the *service-instance feasibility* constraint, evaluated at the service controller, requires run-time querying to evaluate whether an existing vGMux service instance has sufficient capacity/resources, and authenticated to support the new vCPE service for the given subscriber.

These queries place significant burden on the cloud and service controllers, which are primarily responsible for life-cycle management of the cloud and service instances. For instance, a SDN Controller that manages flow control needs to perform topology and configuration operations like injecting routes into the network based on operator specified rules, besides performing periodic management and monitoring activities for the services (*e.g.*, BGP connections). These controllers, which are primarily designed for service life-cycle management as opposed to large-scale query processing, are easily overburdened with a large number of repeated queries coming from the homing service. Hence, network operators typically rate-limit such queries coming from external services. Further, it is impractical for the NSP to re-design these controllers to process larger query volumes since they often depend on third party software. For example, the NSP we worked with built their SDN controller on top of the ODL controller [22].

To quantify the burden of queries at the controllers, we analyzed a week long trace of all homing related queries issued to all cloud and service controllers in a Tier-1 NSP. We make the following observations. First, the homing service would send queries at a rate at least twice the allowed rate for 22% and 44% of the time, to the least and most loaded controllers (*w.r.t.*, homing queries) respectively. To cope with this rate, the controllers queue those queries – resulting in unwanted delays. Second, both cloud and service controllers take a significant amount of time to process and respond to those queries. Figure 2a shows the CDF of the query latency observed at those controllers. As the graph shows, service controllers’ query latency was more than 1 second for 20% of the queries. Third, as a result of both heavy query processing as well as query queuing, solving a single homing request takes a long time. Figure 2b shows the CDF of the total time spent by the controllers per homing request. The graph shows that about 50% of the homing requests required more than 400 seconds of controller’s time across all controllers. With NSPs creating thousands of service instances every day, querying the service and cloud controllers for every new service instance is prohibitively expensive and time-consuming.

We address this challenge through an incremental approach in §IV, where we query instantaneous capacity and other required information during the run-time phase to avoid the wastage of pre-provisioning, but at the same time ensure a far reduced burden of querying at the controllers.

III. StepNet - A COMPOSITIONAL HOMING FRAMEWORK

As described in §II-A, traditional homing approaches require significant changes in the underlying heuristic optimization models when the service composition or homing constraints evolve, because optimization models have tight dependencies across the demands, constraints and objective functions. In this section, we describe how we address this challenge through our novel compositional homing framework, *StepNet*. Through this framework, homing instances of a service can be described through a declarative template that consists of *compositional blocks* which are abstractions that specify demands, candidates, constraints, objective functions, data-sources and heuristic optimization algorithms (terms defined in §I, §II). These abstractions have clearly defined structures, functional behavior and APIs. Our notion of *composability* is that, as long as homing instances are created by mixing and matching these compositional blocks, our incremental approach (see §IV) can provide a solution. This enables service designers to create new homing requests with considerable ease as their customer requirements evolve.

Standardized Compositional Behavior. The composition blocks, specifically the constraints and objective functions, have standardized interfaces and pre-established functional behaviors. For instance, all constraints are designed as plugins exposing a common interface, `solve(homing-context, candidate-set, data-sources)`, where the `homing-context` is an object that captures the current state of the homing request being processed including the demands, constraints, and input parameters. The `candidate-set` is an input set of candidates which are found feasible until the point when the current constraint is invoked. The `data-sources` specify what data sources need to be queried to collect information required to evaluate the constraint. All constraints exhibit a consistent filtering behavior, which eliminates zero or more candidates that do not meet the constraint’s requirements, and returns a subset (not necessarily a strict subset) of the input candidate-set. Similarly, all objective functions expose the interface `compute(optimization-goal, normalization-function, cost-function)`. The `optimization-goal` can be to minimize/maximize, while the `cost-function` can represent different metrics like latency, utilization, dollar costs, *etc.*, while the `normalization-function` allows joint optimization with multiple metrics like latency and utilization by normalizing these values.

Library of Composition Blocks. We distill our detailed study of production services to create a library of composition blocks [23], through which a service designer can obtain common demands, constraints and objective functions. For example, our library contains the candidate types of ‘cloud-region’ and ‘service-slice’ and the associated inventories. The service designer simply needs to mention the VNFs that correspond to these demands. Similarly, the library also contains the constraints for zone and capacity and only the instance specific details like the demands and the actual bandwidth (among others) need to be specified. The library elements for

the objective function and heuristics operate in a similar way.

Listing 1 Structure of a Homing template instance

```

RUN-TIME PARAMETERS:
  instance-id   : 'id of current service instance'
  ↳ being created
  heuristic     : 'Best-fit'
  customer-id   : 'id'
  vpn-key       : 'vpn-k1'
DEMANDS:
  VNF-X:
    cand-type   : 'cloud-region'
    inventory    : 'cloud-controller'
  VNF-Y:
    cand-type   : 'service-slice'
    inventory    : 'network-srvc-provider'
CONSTRAINTS:
  cons-Z:
    demands     : [VNF-X, VNF-Y]
    type        : 'zone'
    zone        : 'cloud-region'
    qualifier    : 'same'
  cons-C:
    demands     : [VNF-Y]
    type        : 'capacity'
    resources    : {'BW' : '1Gbps'}
    subscriber-info: {'id' : customer-id,
                      'key' : vpn-key}
OBJECTIVE-FUNCTION:
  fl:
    f-type      : 'cost'
    demands     : [VNF-X, VNF-Y]

```

Homing Template. To enable service designers to specify various compositional blocks, we provide a declarative template inspired by OpenStack’s Heat template [24], appropriately extended to support the composition blocks described below. Listing 1 shows the structure of a homing specification for the example of the vCPE service described in Figure 1. The `RUN-TIME PARAMETERS` block captures the run-time inputs required for homing, like subscriber information and authentication keys that are typically used to evaluate the constraints (e.g., `vpn-key` is used in `cons-C`). The `DEMANDS` block represents the network elements of the service, the candidate types for each of these elements and the inventory source from which these candidates can be drawn. The `CONSTRAINTS` block lists the constraints, their parameters and the specific *demands* to which the constraint applies. Some constraints are pertinent to a specific demand (e.g., bandwidth capacity required by a VNF), while some constraints span multiple demands (e.g., distance between two VNFs of the service). The `OBJECTIVE-FUNCTION` block specifies the target metric optimization like dollar costs, latency, *etc.* Except for the `DEMANDS` block, the other composition blocks are optional. For instance, some real-world network services do not have any hard constraints, but require optimality of an objective function metric. Finally, the heuristic algorithm that invokes these constraints and objective functions to meet the service homing requirements, can be selected as a part of the composition. For instance, the same homing template instance shown in Listing 1 can be used with either a heuristic best-fit search algorithm or a shortest-path search algorithm².

To illustrate how our compositional framework allows service evolution, consider the example of vCPE service (Fig-

²This template serves as an example and is not intended to be an exhaustive description of all possible composition blocks.

ure 1) providing an added Firewall (vF) function such that the cloud site hosting the vF has sufficient capacity and the vF is co-located with the vG. The vF would be a new *demand* in the DEMANDS block, and the co-location requirement for the vF can be easily added by modifying the *Affinity* constraint to include the vF along with the vG and vGMux. The capacity requirement would be a new *cloud feasibility* constraint added to the CONSTRAINTS block. This compositional framework supports over 1000 instances of production services (see §V) with varying demands and constraints ranging from traditional use-cases like WAN, Private Virtual Networks, and even futuristic network services such as 5G Network Slicing [25].

IV. INCREMENTAL APPROACH TO HOMING

As described in §II-B, traditional homing approaches place substantial burden on the controllers that need to be queried for *run-time* information as part of the homing process. We propose an incremental approach that minimizes the queries made to the controllers while maintaining a reasonable level of solution quality, by leveraging two independent dimensions:

- **Objective-based candidate ranking:** we limit the number of candidates against which the constraints shall be evaluated by incrementally increasing the set of candidates used until a quality solution is found (*i.e.*, we incrementally explore the overall search space of candidates).
- **Cost-based ordered constraint evaluation:** we order constraints such that the least expensive constraints are evaluated first. In doing so, we can more quickly prune out candidates that are not feasible, and only evaluate the most expensive constraints when we know the rest of the constraints are feasible.

In this section, we first describe an overview of the incremental approach (IV-A), and then we expand on the two key concepts that enable it: objective-based candidate ranking (IV-B) and cost-based ordered constraint evaluation (IV-C).

A. Incremental Approach Overview

The incremental approach, in a nutshell, iteratively and carefully expands the search space until it satisfies certain stopping criteria. We highlight our incremental approach in Algorithm 1. The main procedure, FINDSOLUTION, is called for each homing request (an instance of the template presented in Listing 1) and should return a solution (if any). In order to obtain the initial set of potential candidates to pass into FINDSOLUTION for each of the demands, *StepNet* queries multiple data-sources, including the NSP inventory and cloud controllers to determine what candidates can be used by each demand. We treat this as a fixed (minimal) cost for each homing request. To enable the incremental approach, we first rank the candidates (§IV-B) for each of the demands (line 2) in a way that favors “good” candidates that have a higher chance of yielding quality solutions. The second step, which is triggered by the *solution* call in line 11, is to evaluate the constraints in a cost-based order (§IV-C), from most to least expensive, to enable early elimination of candidates.

Algorithm 1 Incremental Algorithm

Input:

```

 $r$  : instance of a homing request consisting of: demands, constraints,
objective function, and a set of initial candidates for each demand
 $p$  : instance of a heuristic algorithm (e.g., best-fit, exhaustive)
 $step$  : incremental step (candidate subset) size
 $tol\_thresh$  : local solution tolerance threshold
1: procedure FINDSOLUTION( $r, p, step, tol\_thresh$ )
2:   RANKCANDIDATES( $r$ )
3:    $best = null$  ▷ keep track of best solution
4:    $tolerance = 0$  ▷ used for stopping_condition
5:   Start with empty available-candidates for each demand
6:   while True do
7:     if stopping_criteria then
8:       break
9:     for each  $d \in r.demands$  do
10:      increment available-candidates[ $d$ ] by  $step$ 
11:       $sol = p.solution(r, available-candidates)$  ▷ triggers
constraints evaluation
12:      if  $sol = null$  then
13:         $step = step * 2$ 
14:        go to 6
15:       $step = \text{original size}$ 
16:      if first solution or  $sol$  is better than  $best$  then
17:         $best \leftarrow sol$ 
18:         $tolerance = 0$ 
19:      else ▷  $sol$  did not improve solution quality
20:         $tolerance ++$ 
21:   return  $best$ 

```

Incremental Search Space Exploration. For each demand, we start with an empty set of *available-candidates* (line 5) – the set of candidates that should be ready for constraint evaluation. In each iteration, then, we increment this set by a *step* (line 10), which is set to some percentage of the whole set of initial candidates (*e.g.*, 2%).

We note that in each iteration, we include the union set of *available-candidates* of all previous iterations plus the current one such that $C_{(i,d)} = C_{(i-1,d)} \cup C'_d$ where $C_{i,d}$ represents the *available-candidates* set for demand d in iteration i and C'_d represents the new candidates added in iteration i . This is necessary since we cannot discard candidates from previous iterations as it will break dependencies imposed by pairwise constraints. Such constraints could force us to choose one candidate for demand X in one iteration, and choose another candidate for demand Y in a different iteration.

Optimizer Independence of the Incremental Approach. After populating the *available-candidates* set for each of the demands in r , we pass r and *available-candidates* to the optimizer instance (line 11) where it evaluates the constraints for those *available-candidates*. By making the optimizer agnostic to how candidates are made available, the network operator can plug in any optimization algorithm (*i.e.*, optimizer instance) without modifying the logic of how candidates should be added to the *available-candidates* set. All the optimizer does is evaluate the constraints for *available-candidates* (triggered by the *solution* call on line 11), and according to its logic, decide what candidates to choose for which demand.

Adaptive Steps. Assigning a small value to the step size (*step*) helps reduce the number of queries to be sent by reducing the number of candidates against which the constraints

are evaluated. However, when there are no feasible candidates in the first few iterations, the incremental algorithm can take a long time to find the first solution. To avoid this, we make the *step* adaptive such that each time the optimizer is not able to find a solution (line 12), we double the *step* size (line 13). When *any* solution is found, we restore the original *step*.

Terminating the Incremental Loop. After getting a solution *sol* (line 11), the incremental algorithm evaluates it, and decides whether to accept it (lines 17-18) or to tolerate it by incrementing the *tolerance* counter (line 20 – when *sol* is not “better” than the best solution so far, by comparing their objective values). The incremental algorithm terminates when certain *stopping_criteria* are met such as when *tolerance* exceeds *tol_thresh*, a tolerance threshold that can be specified by the operator. Another metric that can be incorporated into the stopping criteria is an upper limit on the number of queries that can be made. We describe our stopping criteria in §V.

B. Objective-based Candidate Ranking

Since we incrementally increase the set of candidates available to be evaluated until we find a good solution (calculated on line 11 in Algorithm 1), we can reduce querying if we can find a good solution in as few iterations as possible. This requires that the candidates in early iterations are “good” candidates, so we do not have to proceed to later iterations. To rank the candidates, we leverage the objective function in the following manner. First, we construct a tree where each level in the tree consists of the lists of candidates for a given demand (where each demand is its own level). Each node is connected to the nodes in the next level down (for the next demand), with an edge weight that is set to the added cost of including the given candidate for that demand in the solution according to the objective function. For demands that are not part of the objective function, edge weights are all set to the same value for that demand. A solution path, then, is one candidate from each level in the tree. The rank of the candidate is based on the best solution-path value among each of the solutions paths that cover that candidate.

While the objective function does not indicate whether the candidate is a feasible candidate, it does indicate whether it could be a *good* candidate. We use this to determine how to incrementally increase the set of candidates used in a given iteration. This, in turn, reduces the number of candidates we need to test for feasibility, which is the expensive part of homing. Interestingly, limiting the set of candidates to only those with the best objective values can actually improve the solution quality of the optimization heuristic (see §V), because it increases the chance of selecting a better local optimum.

Note that, objective-based candidate ranking in itself does not incur much query overhead since most services that we found in the Tier-1 NSP do not optimize (*i.e.*, their objective functions) for *run-time* information.

C. Cost-based Ordered Constraint Evaluation

We choose to evaluate constraints in a sequential, ordered manner to eliminate additional queries for candidates once

we know they are infeasible. Unlike *static* constraints (location proximity, zone, *etc*), evaluating *run-time* constraints (instance-feasibility, network latency, *etc*) against a set of candidates triggers queries to be sent to collect *run-time* information for those candidates. Clearly, *static* constraints are the least expensive to evaluate since they do not require queries, and evaluating them first helps prune the set of candidates before getting to evaluate *run-time*, and therefore more expensive constraints. Doing so allows us to evaluate *all* constraints, but against a smaller set of candidates.

When the *static* constraints indicate that a candidate is still feasible, we need to start evaluating the *run-time* constraints. In §II-B, we saw that different queries have different costs, and as such, we still would like to order the constraints. Determining how expensive a *run-time* constraint is to evaluate cannot be directly derived from the constraint properties. So, instead, we propose using past query logs to help calculate a real-time cost for each of the constraints that *StepNet* supports. We assign the rank (or cost) of a constraint as follows: $C(cons) = l * f$, where l and f are the median latency and frequency of queries triggered by constraint *cons* in past logs within a recent time window, respectively. Performing this offline task periodically allows our constraint ranking function to adapt to continuous changes in the network as well as adapt to new constraint types. To bootstrap the process when logs are not available, we assign equal ranks to *run-time* constraints, and after solving a number of homing requests, we re-evaluate these ranks.³

V. EVALUATION

In this section, we validate our key contributions of the compositional homing framework (§III), and the incremental approach (§IV) by answering these two key questions: (i) can we easily compose homing requests for various network services using *StepNet*’s *compositional blocks*? And (ii) does the incremental approach significantly reduce the query cost while retaining quality of solution?

A. Experimental Setup

Implementation We have implemented *StepNet* in Python with $\sim 10k$ LoC. Our current implementation supports 10 types of constraints, 3 optimization heuristics, and 7 objective functions – *all of which are implemented as plugins with APIs that can be easily consumed*. *StepNet* consists of many components (controller, interpreter, optimizer, data router, *etc*) that are designed as microservices that can scale independently. Due to space constraints, we defer the design and implementation of this micro-services architecture to a future paper. This implementation is now being deployed in the production network of a tier-1 NSP.

Trace-driven Evaluation. We collected 7 consecutive days of longitudinal homing traces from the production logs of a tier-1 NSP, which includes the homing requests, query parameters, query responses and latency observed for the

³We note that our incremental approach is independent of and orthogonal to the constraint type (be it quantitative or qualitative), and only affects the number of candidates being processed by the constraint.

queries sent to the different controllers. We used data collected from the trace to comprehensively emulate various aspects of a homing service running in production by following the same distribution found in the trace for the following aspects: feasibility/capacity responses (*i.e.*, whether a cloud site has enough capacity to home a given demand), query latency, geographical location density (*i.e.*, how many cloud sites per country). With our emulation framework, we emulate a total of 2400 cloud sites and service instances (*i.e.*, potential candidates), enabling us to evaluate *StepNet* at a large scale.

B. Supporting Flexible Service Composition

The traditional approach to homing, where new optimization models and heuristics are designed for each new service offering, incurs significant effort that runs counter to the requirements of maintainability and evolvability of services in NSP infrastructures (§II). One of the main goals of *StepNet* is to easily accommodate new network services by allowing service designers to compose homing requests for such services by mixing and matching different constraints and objective functions through our compositional framework.

Easy Service Composition. To demonstrate *StepNet*’s ability to accommodate various services, we obtained 12 service models from a tier-1 NSP, ranging from simple ones (1 demand, 1 constraint) to more complex ones (6 demands, 42 constraints), covering single-homed as well as multi-homed services. In a few days, the service designers could generate the homing requests for those services using the template in Listing 1. Further, for each of the 12 services, then, we generate 100 instances that have different *run-time* parameters (required capacity, customer location, *etc.*), while ensuring that we retain the same *service model* including demands, constraints, and constraint types as observed in production. This enabled us to create 1200 homing request instances with which we can test the incremental approach.

Supporting Service Evolution. Our base implementation of *StepNet* could support all the constraints required for the 12 services. However, a *run-time* objective function (minimize resource utilization) required by one of the services was not originally part of our implementation. With only 78 lines of code, we were able to implement this function along with the necessary data hooks to query cloud controllers for resource utilization information. This highlights the ease with which *StepNet* can adapt to evolving service/business requirements.

Supporting Pluggable Optimization Heuristics. Since our incremental approach is decoupled from the specific optimization heuristic, contrary to existing solutions that design a service-specific heuristic for each use-case/service, we can plug-in a variety of optimization heuristics through our compositional framework (line 11 in Algorithm 1). To demonstrate this aspect, we wrote two basic heuristic algorithms: *random* and *shortest-path first (SPF)*, as well as a state-of-the-art optimization heuristic: backtracking best-fit (*BACBF*).

The *Random* heuristic randomly assigns a feasible *candidate* to a given *demand*. The *SPF* heuristic, on the other hand, is more comprehensive. It exhaustively identifies the best

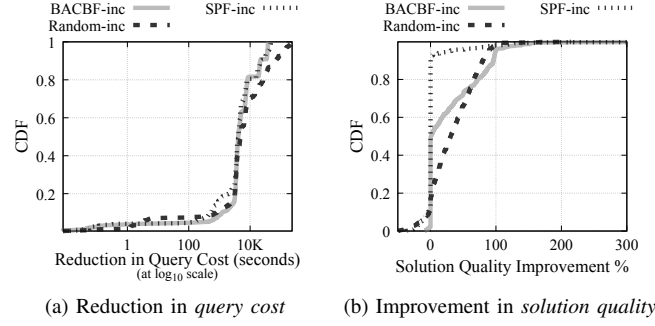


Fig. 3: Results of running 1200 homing requests. This shows that the incremental approach not only reduces query cost (left) for all three heuristics, but it does so while improving solution quality (right).

candidate-demand mappings that optimize the objective value based on an implementation of Dijkstra’s algorithm [26]. The *BACBF* heuristic is a modified version of the best-fit optimization heuristic augmented with a backtracking ability. Best-fit has proven very effective in recent placement optimization studies [8], [27], [28]. For details on how we implement each of these optimization heuristics we refer the reader to [29].

C. Reducing Query Cost with the Incremental Approach

We evaluate *StepNet*’s incremental approach to answer the following questions: (1) how much cost is the incremental approach able to reduce? (2) does the incremental approach impact solution quality (*i.e.*, objective value)? (3) is there a benefit to objective-based candidate ranking? and (4) is there a benefit to cost-based constraint ordering?

To answer these questions, we ran a set of experiments evaluating *StepNet* on CloudLab [30]. We use the three heuristics described in §V-B with two configurations: with the incremental approach described in §IV (where we let the incremental approach decide what candidates the heuristic should see – denoted with *inc*) and without it (where we pass all potential candidates to the heuristic – denoted with *all*). Doing so yields six heuristic variants: *Random-all*, *Random-inc*, *SPF-all*, *SPF-inc*, *BACBF-all*, and *BACBF-inc*. Since the *SPF-all* configuration is the most comprehensive, we treat its solution for a given homing request as a baseline to measure how good the solutions of other heuristic configurations are. Moreover, we timeout *SPF-all* (and for that matter all heuristics) after 30 minutes of processing time, and use the best solution it could calculate at that point.

Incremental algorithm parameters. For the incremental versions of the heuristics, we set the step size (*step*) to 2% of the initial candidate set (while enabling adaptive steps), and we used a minimum of 5% improvement of any two successive solutions as our tolerance threshold (*tol_thresh*). We also used a relaxed *stopping_criteria* throughout our experiments that terminates the incremental algorithm if either: *tolerance* > 1, processing timeout (30 minutes) is reached, or the total number of queries exceeds 2K queries. We generated these parameters by experimenting with a range of values and selected ones that yielded the best outcome. We note that we

enable ordered constraint ranking for both incremental and non-incremental versions in our evaluation.

How well does the incremental approach work?

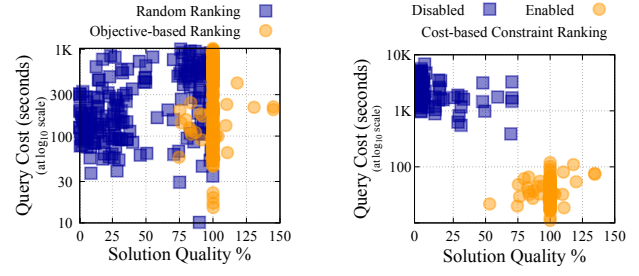
We ran experiments with the 1200 homing requests, that we generated for the 12 services described earlier, through all six configurations described above. For each homing request, we measure two key metrics: *query cost* and *solution quality*. The *query cost* is the cumulative time spent by all data sources in responding to queries issued for a given homing request, which represents the “load” placed on the corresponding controllers. The *solution quality* is a percentage value that shows how close a given heuristic’s objective value to the baseline’s (*SPF*-all) objective value. A value greater than 100% indicates that a solution is better than the baseline.

Does the incremental approach reduce *query cost*? Figure 3a shows reduction in *query cost* with the incremental approach over the non-incremental approach for all the heuristics. The X-Axis shows the reduction in *query cost*, while the Y-Axis shows the CDF of the fraction of 1200 homing requests. From the graph, we make several observations. First, the incremental approach reduces the *query cost* across all three heuristics, highlighting the benefits of our incremental approach. Next, the incremental approach was able to reduce the *query cost* when compared to the non-incremental heuristics by more than 1K seconds for about 80% of homing requests (and by more than 10K seconds for 20% of homing requests). Finally, the incremental approach was able to (not shown in Figure 3a) reduce 60% of the *query cost* for 80% (and 95% for 20%) of the homing requests.

Does the incremental approach impact *solution quality*?

Figure 3b shows the CDF of the percentage improvement in *solution quality* with the incremental approach over their non-incremental counterparts for all the three heuristics. The incremental approach improves the *solution quality* for the majority of cases for the *BACBF* (50%) and *Random* (70%) heuristics, and provides comparable *solution quality* for the *SPF* heuristic. The benefits primarily come from the incremental algorithm’s ability to intelligently rank candidates by favoring those that could lower the objective value (for minimizing objective functions) while limiting the search space. This, in turn, helps the heuristics terminate at a better local optimum. For a small fraction of homing requests, the incremental approach degrades the *solution quality* by at most 43% due to its limiting the candidate search space. We argue that this is a reasonable trade-off especially when considering the substantial reductions in *query cost*. Henceforth, due to space constraints, we only show results for the *BACBF* heuristic, since it consistently outperforms the others.

Objective-based vs. random candidate ranking. We now compare *BACBF-inc* for 100 homing requests for the same service with two configurations of candidate ranking: (1) random candidate ranking, and (2) objective-based candidate ranking. Figure 4a shows a scatter plot with the *solution quality* along the X-Axis and *query cost* along the Y-Axis. A value greater than 100% for the *solution quality* indicates a better solution,



(a) Objective-based candidate ranking vs. random ranking

(b) Enabling/disabling cost-based constraint ranking

Fig. 4: Key features of the incremental approach (using *BACBF-inc*): objective-based candidate ranking (a) and cost-based constraint ranking (b), provide significant benefit to *solution quality* and *query cost*.

while a value less than 100% indicates a degradation in the *solution quality*. As observed from the graph, while objective-based ranking does not provide significant value in terms of *query cost*, it provides significantly better (3x for 50% of requests) *solution quality* (X-Axis).

Benefits of cost-based constraint ordering. Finally, we run 100 homing requests through two configurations of the *BACBF-inc* heuristic: (i) with the cost-based constraint ranking and (ii) without constraint ranking. We select one of the 12 services that had the most number of constraints – 42 constraints spread across 6 demands of the service. To highlight the penalty of incorrect ranking, we reversed the ordering of constraints for the latter configuration (ii). Doing so provides us with a lower bound for parallel constraint evaluation (*i.e.*, the same candidate set is evaluated by all constraints).

Figure 4b shows a scatter plot with the *query cost* along the Y-Axis and *solution quality* along the X-Axis. We observe that the *query cost* increases by at least one order of magnitude, when cost-based constraint ranking is disabled. Further, this leads to significant degradation in *solution quality* (higher objective values). In theory, constraint-ordering should not affect the final outcome of an algorithm and hence the quality of the solution. However, in our experiments, evaluating query-intensive constraints (*i.e.*, with cost-based ranking disabled) in the initial stages of constraint evaluation causes the incremental algorithm to hit the time-out limit of 30 minutes.

VI. CONCLUSION

Homing of VNFs is a crucial part of the life-cycle management of complex network services. Traditional approaches to homing adopted by NSPs are service-specific and do not easily accommodate service evolution, critically affecting the maintainability of these services. Further, homing involves burdensome querying of several cloud, SDN and service controllers, the query cost of which needs to be limited. Our compositional homing framework, *StepNet*, was designed to solve these challenges, catering to a growing number of services. Our results show that our incremental approach is able to provide good quality solutions, while reducing query costs by 92% for half of the homing requests when compared to non-incremental approaches.

REFERENCES

- [1] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, "Network function virtualization: Challenges and opportunities for innovations," *IEEE Communications Magazine*, vol. 53, no. 2, pp. 90–97, 2015.
- [2] G. Jung, M. A. Hiltunen, K. R. Joshi, R. K. Panta, and R. D. Schlichting, "Ostro: Scalable placement optimization of complex application topologies in large-scale data centers," in *35th IEEE International Conference on Distributed Computing Systems, ICDCS 2015, Columbus, OH, USA, June 29 - July 2, 2015*, 2015, pp. 143–152. [Online]. Available: <https://doi.org/10.1109/ICDCS.2015.23>
- [3] X. Meng, V. Pappas, and L. Zhang, "Improving the scalability of data center networks with traffic-aware virtual machine placement," in *INFOCOM, 2010 Proceedings IEEE*. IEEE, 2010, pp. 1–9.
- [4] J. T. Piao and J. Yan, "A network-aware virtual machine placement and migration approach in cloud computing," in *Grid and Cooperative Computing (GCC), 2010 9th International Conference on*. IEEE, 2010, pp. 87–92.
- [5] W. Fang, X. Liang, S. Li, L. Chiaraviglio, and N. Xiong, "Vmplanner: Optimizing virtual machine placement and traffic flow routing to reduce network power costs in cloud data centers," *Computer Networks*, vol. 57, no. 1, pp. 179–196, 2013.
- [6] K. Le, R. Bianchini, J. Zhang, Y. Jaluria, J. Meng, and T. D. Nguyen, "Reducing electricity cost through virtual machine placement in high performance computing clouds," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, p. 22.
- [7] V. Shrivastava, P. Zerfos, K.-W. Lee, H. Jamjoom, Y.-H. Liu, and S. Banerjee, "Application-aware virtual machine migration in data centers," in *INFOCOM, 2011 Proceedings IEEE*. IEEE, 2011, pp. 66–70.
- [8] M. Xia, M. Shirazipour, Y. Zhang, H. Green, and A. Takacs, "Network function placement for nfV chaining in packet/optical datacenters," *Journal of Lightwave Technology*, vol. 33, no. 8, pp. 1565–1570, 2015.
- [9] H. Moens and F. De Turck, "Vnf-p: A model for efficient placement of virtualized network functions," in *10th International Conference on Network and Service Management (CNSM)*, 2014, pp. 418–423.
- [10] M. Ghaznavi, A. Khan, N. Shahriar, K. Alsubhi, R. Ahmed, and R. Boutaba, "Elastic virtual network function placement," in *Cloud Networking (CloudNet), 2015 IEEE 4th International Conference on*. IEEE, 2015, pp. 255–260.
- [11] A. Patel, M. Vutukuru, and D. Krishnaswamy, "Mobility-aware vnf placement in the lte epc," in *2017 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE, 2017, pp. 1–7.
- [12] K. M. Bretthauer, "Capacity planning in manufacturing and computer networks," *European Journal of Operational Research*, vol. 91, no. 2, pp. 386 – 394, 1996. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/0377221794003025>
- [13] R. Z. Farahani, N. Asgari, N. Heidari, M. Hosseini, and M. Goh, "Survey: Covering problems in facility location: A review," *Comput. Ind. Eng.*, vol. 62, no. 1, pp. 368–407, Feb. 2012. [Online]. Available: <http://dx.doi.org/10.1016/j.cie.2011.08.020>
- [14] R. Z. Farahani, M. SteadieSeifi, and N. Asgari, "Multiple criteria facility location problems: A survey," *Applied Mathematical Modelling*, vol. 34, no. 7, pp. 1689 – 1709, 2010. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0307904X09003242>
- [15] R. Mayer, H. Gupta, E. Saurez, and U. Ramachandran, "Fogstore: Toward a distributed data store for fog computing," in *2017 IEEE Fog World Congress (FWC)*. IEEE, 2017, pp. 1–6.
- [16] M. Karlsson and C. Karamanolis, "Choosing replica placement heuristics for wide-area systems," in *24th International Conference on Distributed Computing Systems, 2004. Proceedings*. IEEE, 2004, pp. 350–359.
- [17] D. Harutyunyan, N. Shahriar, R. Boutaba, and R. Riggio, "Latency-aware service function chain placement in 5g mobile networks," in *2019 IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 2019, pp. 133–141.
- [18] N. Kodiaev, S. Bayless, F. Ruffy, I. Beschastnikh, H. H. Hoos, and A. J. Hu, "Vnf chain allocation and management at data center scale," in *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems*. ACM, 2018, pp. 125–140.
- [19] V. Heorhiadi, M. K. Reiter, and V. Sekar, "Simplifying software-defined network optimization using sol," in *NSDI*, 2016, pp. 223–237.
- [20] V. Heorhiadi, S. Chandrasekaran, M. K. Reiter, and V. Sekar, "Intent-driven composition of resource-management sdn applications," in *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, 2018, pp. 86–97.
- [21] IBM, "Ibm ilog cplex optimization," 2020. [Online]. Available: <http://www.cplex.com/>
- [22] OpenDaylight, "Opendaylight," 2020. [Online]. Available: <https://www.opendaylight.org/>
- [23] ONAP, "Homing specification guide," <https://wiki.onap.org/display/DW/OOF-HAS+Homing+Specification+Guide>, 2020.
- [24] Openstack, "Openstack heat orchestration template," 2020. [Online]. Available: https://docs.openstack.org/heat/rocky/template_guide/hot_guide.html
- [25] F. T. A. Council, "5g network slicing whitepaper," 2018. [Online]. Available: <https://transition.fcc.gov/bureaus/oet/tac/tacdocs/reports/2018/5G-Network-Slicing-Whitepaper-Finalv80.pdf>
- [26] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.
- [27] J. Dong, X. Jin, H. Wang, Y. Li, P. Zhang, and S. Cheng, "Energy-saving virtual machine placement in cloud data centers," in *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. IEEE, 2013, pp. 618–624.
- [28] M. Ghobaei-Arani, M. Shamsi, and A. A. Rahmanian, "An efficient approach for improving virtual machine placement in cloud computing environment," *Journal of Experimental & Theoretical Artificial Intelligence*, vol. 29, no. 6, pp. 1149–1171, 2017.
- [29] A. Alsudais, S. P. Narayanan, B. Balasubramanian, Z. Huang, and E. Keller, "Iterative plugin design and implementation," 2020. [Online]. Available: <https://wiki.onap.org/display/DW/Iterative+Plugin+Design+and+Implementation>
- [30] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra, "The design and operation of CloudLab," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, Jul. 2019, pp. 1–14. [Online]. Available: <https://www.flux.utah.edu/paper/duplyakin-atc19>