Dependency-Based Evolution Planning of a Multi-Version Microservice RAN Application

Emma Witt[1] and Simin Nadjm-Tehrani[2][0000−0002−1485−0802]

[1] Linköping University, Linköping, Sweden **
emmalou.witt@gmail.com
[2] Linköping University, Linköping, Sweden
simin.nadjm-tehrani@liu.se

Abstract. Future communication systems are complex infrastructures with virtualized software services that are updated regularly as requirements evolve. This paper addresses the adaptation of microservices under dependency and efficiency constraints. We formalize the evolution of microservices as a combination of two subproblems, the Microservice Dependency Problem and the Evolution Planning Problem, both of which are difficult to solve optimally. We then propose a method based on the Binate Covering Problem (BCP) with branch-and-bound, and introduce a novel algorithm that finds the deployment steps towards a desired new configuration. Our proposed method, DEP-DS, is then compared with two heuristics on three sample datasets from historical radio access network update records run on Kubernetes. We further show that BCP with greedy search is faster but finds fewer solutions to the evolution plan. Overall, DEP-DS is able to find solutions to all samples, generates deployment plans within an average time of 1-2 seconds, and the plans are similar to other heuristics in terms of CPU usage.

Keywords: Microservices · Evolution plans · RAN virtualization

1 Introduction

A Radio Access Network (RAN) manages wireless communication between user devices and the core network. As RAN software grows in complexity, microservice architectures have been adopted to split systems into independently executable components. Despite low coupling, microservices often require integration to deliver full functionality, resulting in complex and hard-to-manage dependencies in large systems. The version dependency problem arises from the independent deployment of microservices. When a microservice is upgraded to a newer version, its communication endpoints may change, and other microservices and external systems that previously interacted with it will no longer be able to do so. To ensure smooth transitions, multiple versions often coexist. A multi-versioning

** The work was carried out when the first author **was at** Ericsson AB, Linköping. The work does **not relate** to the first author's current position at Amazon Web Services.

strategy [8] or dependency-based orchestration [1] helps manage these deployments. The goal is to deploy and maintain compatible versions without service disruption or resource waste. Replacing a needed microservice as a result of an update can make other service chains non-operational. Maintaining multiple versions in parallel to preserve all existing dependencies intact will inevitably waste resources. Manual deployment is error-prone, especially in complex systems like RAN where there may exist 10 or more distinct microservices and multiple versions of them at any point in time, making automation critical. However, algorithms to perform the above task autonomously have a large state space (of all service chains, all possible versions, and endpoints) to deal with. They still need to perform updates with reasonable latency and strike a balance between (CPU) resource utilization and service continuity.

This paper addresses this challenging problem by subdividing it into two orthogonal problems for each update cycle. The first problem is to create a representation of current dependencies that need to be maintained after a revision of a bunch of microservices, the Microservice Dependency Problem (MDP). The second is to construct a deployment plan for desired updates across all microservice chains, the Evolution Planning Problem (EPP). Our study of the existing literature reveals that solutions to the first problem exist; however, to our knowledge, the second problem has not been characterized or solved in the context of dependency-constrained settings, nor does an existing approach appear to address both problems jointly. An ambitious solution to the problem would be to optimize the solution for multiple requirements, e.g., maximizing the number of performed updates at each cycle, minimizing the use of CPU (thereby energy) after the updates, minimizing the number of versions for each microservice, adding constraints as to how fast an update round should be, and so on. However, solving the multi-objective optimization problem would have an infeasible overhead in RANs due to the combinatorial problems mentioned before.

This paper is a first attempt at finding a combined solution to the above two problems through heuristics that can be studied in an experimental setting using realistic data. Our method leverages solutions to two algorithms, namely the Binate Covering Problem (BCP) solved using Branch and Bound (BB) resolution to achieve the combined goal, together with a tree-based algorithm to find the path to the desired deployment from the current deployment. The approach uses BCP as the vehicle to find the desired post-deployment state via the Branch and Bound algorithm. The practical applicability of the approach is then evaluated with historical data from a real RAN system. The contributions of this paper are as follows:

- Formally defining the microservice dependency problem, MDP, and a heuristic for the evolution planning problem, EPP, in RAN applications.
- Proposing a Dependency-based Evolution Planning and Deployment Solver (DEP-DS) method using BCP, BB, combined with a novel algorithm, and implementing it in a Kubernetes environment, where Kubernetes is a well-known platform for virtualized deployments.

– Extracting representative samples based on historical data from real RAN application updates to use as a basis for evaluation of our method, and make them available to other researchers.
– Evaluating DEP-DS in terms of finding an evolution plan for the extracted samples, average time to compute, and CPU usage in planned evolutions compared to two baselines.

The structure of the paper is as follows. Section 2 describes the required background and the relation to previous work. Section 3 presents the problem formalization and assumptions that capture our system model. Section 4 presents the algorithms that make up the DEP-DS method for constructing evolution plans, and Section 5 evaluates it. The paper is concluded in Section 6.

2 Background and Earlier Work

This section reviews previous research on microservices orchestration and relates it to our problem area and approach. We first review the works that address the main goal, namely, the evolution planning strategy. Then we relate the problem of constructing the dependency graph for microservice updates to known graph construction and manipulation problems. Finally, we describe the necessary background on BCP and BB.

2.1 Earlier work

Although the problem of updating multiple nodes in a distributed system with consistency requirements is an old problem in computer science, the deployment of virtualized services in networks makes the problem more complex [10]. Researchers have recognized that careful analysis of update algorithms is critical to preventing failures in systems with high availability requirements [11]. The problem is multi-faceted in the sense that high-availability systems need to have several versions of each service running in parallel, meaning that a snapshot of the system does not have a fixed number of nodes as in the classical problem.

Moreover, in addition to the functional correctness of individual updates, the overall update process has other dimensions: the practical interoperability of services in multiple programming languages [12], the amount of resources used before and after updates, and the efficiency of the update process itself [13], or the traceability of updates and return to a pre-update state[14].

Works that focus on evolution planning with some resource constraints typically end up with scalability problems as optimization within large state spaces is not feasible in continuous evolution [13], for example, evaluate a greedy method on 6 worker nodes and 11 containers. In this paper, we focus on the process of creating evolution plans with heuristics to manage the run-time scalability issue and do not aim at finding optimum solutions.

He et al. [2] proposed a greedy-based algorithm to generate an evolution plan that minimizes average response time while adhering to resource constraints.

This approach prioritizes immediate gains, similar to the dependency-based orchestration algorithms explored in this thesis, which consider the sequence of resource utilization and deployment operations. Such algorithms require a thorough search across potential configurations to identify optimal solutions, particularly for complex microservice interactions that suffer from the combinatorial explosion.

The microservice dependency problem parallels well-known problems in software package management, where the objective is to assign specific package versions that satisfy dependency constraints without conflicts [7]. Unlike traditional package managers, microservice architecture allows multiple versions to be concurrently active, which adds complexity to dependency resolution.

The Minimum Set Cover Problem (MSCP) is a special case of the package management problem, which is solved using various heuristic approaches, such as the Hill Climbing algorithm proposed by Akhter [3]. MSCP is based on a predefined universe of constraints, and solving it involves finding a minimum number of sets that cover all elements in it. In contrast, in our case, it is unclear which microservices should be included in the deployment, making the universe of constraints unnecessarily large and the solution thus likely to be suboptimal.

The package management problem can be modeled as a Boolean satisfiability (SAT) problem, where packages are represented as Boolean variables and constraints (dependencies/conflicts) are described as clauses, which together form a Boolean formula in Conjunctive Normal Form (CNF). Once encoded, SAT solvers can be used to determine whether a feasible set of packages exists that satisfies each clause in the CNF [5]. While the SAT focuses purely on feasibility, the Binate Covering Problem (described below) extends this by introducing a cost minimization objective (e.g., minimizing the number of selected variables or weighted variables). The SAT problem is NP-complete, and BCP is known to be at least as hard as SAT because any instance of SAT can be reduced to an equivalent instance of BCP.

2.2 Background

The binate covering problem (BCP) is a combinatorial optimization problem that seeks to choose a minimum-cost assignment of truth values (0/1) to Boolean variables, or *literals*, that satisfies a collection of clauses. Clauses can contain complemented and uncomplemented literals. BCP is usually represented by a binate matrix $\mathcal{A} \in \{-, 0, 1\}^{m \times n}$, with m rows representing clauses (constraints) and n columns representing Boolean variables, and each entry A_{ij} is defined as:

$$A_{ij} = \begin{cases} 1, & \text{if} \quad \text{variable } c_j \text{ appears in clause } r_i \\ 0, & \text{if} \quad \text{variable } c_j \text{ appears in in complemented form in clause } r_i \\ -, & \text{otherwise} \end{cases} \tag{1}$$

A variable assignment means choosing a truth value for each variable: $x_j = 1$ if the variable c_j is selected (set to true) or $x_j = 0$ if variable c_j is not selected (set to false). An assignment x_j covers clause r_i if it satisfies the clause according to the matrix, i.e., $x_j = 1$ when $A_{ij} = 1$ or $x_j = 0$ when $A_{ij} = 0$. Given that each variable c_j is associated with a cost w_j, the objective of the BCP is to find a variable assignment x such that all rows are covered and the total cost is minimized: $\min \sum_{j=1}^{n} w_j x_j$ [4].

Solving the BCP typically starts with a preprocessing (reduction) phase before the actual search. If a row i can only be covered by one variable c_j, then that variable is considered essential and must be assigned A_{ij} in all feasible solutions. Furthermore, a row r_k is dominated by another row r_l if every variable in r_l is also present in r_k, with the same sign, ie, satisfying r_l automatically satisfies r_k. Removing dominated rows and all rows covered by essential variables simplifies the matrix without losing potential solutions. Iterative approaches of such reductions, for example, Gimpel's reduction, produce a reduced covering matrix. If the reduced matrix ends up empty, a minimal and immediate solution can be obtained from the essential variables identified.

Suppose such a minimal and immediate solution cannot be obtained. In that case, Branch and Bound (BB) resolution can be applied, where the problem is partitioned into subproblems (branches), each of which is attempted to be solved recursively to the optimal level [6]. For each subproblem, we approximate a lower bound L on the objective value. If the current best solution is greater than or equal to L, we conclude that this branch cannot improve upon the current best solution and prune it. Finding the exact lower bound is as difficult as solving the BCP itself and typically requires heuristic methods. Coudert [4] utilizes the concept of a Maximal Independent Set (MIS), which is the largest subset of rows such that no two rows cover the same column; it is maximal because no additional rows can be included without violating this independence. The greedy procedure to obtain the MIS is as follows: First, all rows with negated variables are moved. Then, the shortest row is iteratively added to MIS (the row length is given by $|r_i| = \sum_{j=1}^{n} \mathbf{1}_{\{A_{i,j} \in \{0,1\}\}}$, ie, the sum of all non-empty entries in that row), is iteratively added to MIS. This process is repeated until all rows have been removed. Finally, the lower bound is given by the sum of the weights of each row $r_i \in MIS$, that is, the cost of the least costly variable in MIS:

$$L = \sum_{r_i \in MIS} \text{weight}(r_i) = \sum_{r_i \in MIS} \min_{j \in \{j | A_{ij}=1\}} w_j \tag{2}$$

The heuristic for selecting variable c_j^* in branching considers columns covering many rows and is less costly compared to the weights of these rows, according to Equation 3. Short rows have fewer available columns, making decisions around them more impactful. Columns intersecting many short rows are thus favored to prioritize solving the most constrained of the problem first [4].

$$c_j^* = \underset{j \in 1,\ldots,n}{\arg\max} \left(\frac{1}{w_j} \sum_{r_i \in \{j | A_{ij}=1\}} \frac{\text{weight}(r_i)}{|r_i|} \right) \tag{3}$$

3 System model and Problem definition

We now formally define our problem and present the assumed model. The microservice version dependency problem can be formulated as a binary optimization problem over a set of versioned microservices and versioned interfaces. The dependencies between microservices are captured by their provided (and consumed) interfaces to (provided by) other microservices.

Definition 1: Microservice Dependency Problem Let $S = \{S_1, S_2, \ldots, S_m\}$ be the set of distinct microservices and $V_j = \{S_j^{(1)}, S_j^{(2)}, \ldots, S_j^{(k_j)}\}$ the set of available versions of S_j, where $S_j^{(i)}$ denotes version i of microservice S_j. The universe of deployable microservice versions can thus be defined as:

$$M = \bigcup_{j=1}^{m} V_j = \left\{ S_j^{(i)} \,\middle|\, 1 \leq j \leq m,\ 1 \leq i \leq k_j \right\}.$$

Each versioned microservice $S_j^{(i)} \in M$ is associated with a set of provided interfaces $I_p(S_j^{(i)}) \subseteq I$, a set of consumed interfaces $I_c(S_j^{(i)}) \subseteq I$, and a CPU requirement $\text{cpu}(S_j^{(i)}) \in \mathbb{R}^+$, where I denotes the universe of versioned interfaces.

We define the final deployment set as:

$$D_f = \{S_j^{(i)} \in M \mid x_j^{(i)} = 1\}, \tag{4}$$

$$\text{where} \quad x_j^{(i)} = \begin{cases} 1, & \text{if microservice version } S_j^{(i)} \text{ is deployed.} \\ 0, & \text{otherwise.} \end{cases} \tag{5}$$

Constraints Each deployment set must satisfy the following constraints:

$$\sum_{S_j^{(i)} \in M : v \in I_p(S_j^{(i)})} x_j^{(i)} \geq 1, \qquad \forall v \in \mathcal{E}_{\text{req}} \tag{C1}$$

Given a set of required external interfaces $\mathcal{E}_{\text{req}}$, *the external interface constraint* (C1) ensues that each $v \in \mathcal{E}_{\text{req}}$ is provided by at least one deployed microservice.

$$x_j^{(i)} \leq \sum_{S_{j'}^{(i')} \in M : v \in I_p(S_{j'}^{(i')})} x_{j'}^{(i')}, \quad \forall S_j^{(i)} \in \mathcal{M},\ \forall v \in I_c(S_j^{(i)}) \tag{C2}$$

The internal interface dependency constraint (C2) ensures a microservice S_j^i can only be deployed if each interface it consumes is provided by at least one deployed microservice.

$$\text{CPU}_{\text{total}} = \sum_{S_j^{(i)} \in \mathcal{D}_f} x_j^{(i)} \cdot \text{cpu}(S_j^{(i)}) \leq \text{CPU}_{\max} \tag{C3}$$

The resource constraint (C3) ensures the deployment stays below a resource quota $\text{CPU}_{\max}$.

Definition 2. Evolution Planning Problem The evolution planning problem involves transitioning a microservice-based system from an initial deployment state $D_0 \subseteq M$ to a target deployment state D_f. This transition is governed by an Evolution Plan EP, formally represented as:

$$EP = \langle DS_1, \ldots, DS_f \rangle, \tag{6}$$

where each deployment step DS_k is a sequence of operations: (7)

$$DS_k = \{o_{k_1}, \ldots, o_{k_K}\}, \quad \text{and each operation is either} \tag{8}$$

$$o_k = \begin{cases} \text{Deploy}(S_j^{(i)}), & \text{deploying microservice version } S_j^{(i)}. \\ \text{Remove}(S_j^{(i)}), & \text{removing an existing microservice version } S_j^i \end{cases} \tag{9}$$

Applying all operations in DS_k updates the system incrementally:

$$D_k = (D_{k-1} \cup \text{deploys}(DS_k)) \setminus \text{removes}(DS_k), \quad \text{where} \tag{10}$$

$$\text{deploys}(DS_k) = \{S_j^{(i)} \mid o_k = \text{Deploy}(S_j^{(i)}),\ o_k \in DS_k\ \} \tag{11}$$

$$\text{removes}(DS_k) = \{S_j^{(i)} \mid o_k = \text{Remove}(S_j^{(i)}),\ o_k \in DS_k\ \} \tag{12}$$

Each state of the system D_k represents a valid subset of microservice versions deployed that satisfies predefined constraints: $D_k \vDash C1, C2, C3$,

Objective The deployment set D_f is not necessarily unique - multiple allowed deployment sets may exist, as well as numerous EPs may exist between D_0 and the same final state D_f. The objective is to determine our preferred final deployment state D_f and compute EP from our current deployment state to D_0 to D_f so that the total usage of CPU resources is minimized, reflecting the goal: $\min(\text{CPU}_{\text{total}})$.

4 Our proposed approach: DEP-DS

This section describes the main contribution of this work. The Dependency-based Evolution Planning and Deployment Solver, DEP-DS, aims to solve the microservice dependency problem and the evolution planning problem in two stages:

- Solving the Microservices Dependency Problem: Given a universal set of deployable microservices M, a current deployment set D_0, find a final deployment set D_f such that all external interface requirements $\mathcal{E}_{req}$, internal dependencies are satisfied while the overall system remains below a resource quota CPU_{max}. The objective is to minimize the CPU usage of D_f, given that each microservice is associated with a $\text{cpu}(S_j^{(i)})$. MDP is modeled as a BCP and solved by utilizing branch and bound.
- Solving the Evolution Planning Problem: Given the calculated output D_f, find the Evolution Plan EP consisting of a sequence of deployment steps (deployments and removals) to transition from the current state to D_f.

4.1 Solving the Microservice Dependency Problem

The microservice version dependency problem is addressed by first modeling it as a Binate Covering Problem (BCP) and solving it using Branch and Bound resolution. Each microservice version $S_j^{(i)} \in M$ corresponds to a Boolean variable $x_j^{(i)}$, representing the inclusion of a microservice version, which is associated with a column in the covering matrix. Furthermore, each microservice version $S_j^{(i)}$ is associated with a cost $= c_j^{(i)}$ (e.g., CPU resource usage). Internal and external interface constraints are expressed as Boolean clauses over these variables and thereby define clauses in the covering matrix. External interface constraints (C1) yield positive clauses 13 and internal interface requirements yield implications. Internal interface dependencies (C2) imply a relationship between selected microservices and the requirement that at least one provider for each consumed interface must be chosen. Applying the conversion rule: $P \implies Q \leftrightarrow \neg P \vee Q$, this translates into a Boolean clause with one negated literal and at least one positive literal 14 [9].

$$\vee_{S_j^{(i)} \in M : v \in I_p(S_j^{(i)})} x_j^{(i)} \quad \text{for all} \quad v \in \mathcal{E}_{\text{req}} \tag{13}$$

$$\neg x_j^{(i)} \vee \left(\vee_{S_{j'}^{(i')} \in M : v \in I_p(S_{j'}^{(i')})} x_{j'}^{(i')} \right), \forall S_j^{(i)} \in \mathcal{M},\ \forall v \in I_c(S_j^{(i)}) \tag{14}$$

We calculate the lower bound L as the sum of the costs of all variables covering a row in the Maximal Independent Set (MIS): $L = \sum_{r_k \in MIS} \sum_{S_j^{(i)} \in R_{r_k}} c_j^{(i)}$, where $R_{r_k} = \{S_j^{(i)} \in M | A_{k,S_j^{(i)}} = 1\}$. This differs slightly from Equation 2, where only the least costly variables of each row in the MIS were added to the lower bound. Since no complemented rows, i.e., internal dependencies, are included in MIS, we are likely to get a lower bound far from the actual lower bound using Equation 2. Thus, this decision leads to faster convergence. We adapt the heuristic selection for case splitting as explained by Coudert [4] (Equation 3) to select microservice $S_{j^*}^{(i^*)}$ to branch on. Redefining the variables to represent our system's resources and dependency relationships gives the following:

$$S_{j^*}^{(i^*)} = \underset{S_j^{(i)} \in M}{\arg\max} \left(\frac{1}{c_j^{(i)}} \sum_{r_k \in R_{S_j^{(i)}}} \frac{\text{weight}(r_k)}{|r_k|} \right), \tag{15}$$

where

$$R_{S_j^{(i)}} = \{r_k \in M | A_{k,S_j^{(i)}}\}$$

4.2 Solving the Evolution Planning Problem

Algorithm 1 computes a valid Evolution Plan EP for transitioning a microservice from a current deployment state D_0 to a target deployment state D_f, where

each deployment state $D_t \in \{D_0, D_1, \ldots, D_{f-1}, D_f\}$ is reached by applying a sequence of deployment operations according to Equation 6. This process begins by identifying the required changes, specifically the microservices *to_add* and *to_remove* by comparing D_{curr} with D_f. This is preceded by identifying the allowed Add and Remove operations to transition from D_{curr} to D_{next}. The algorithm is recursively applied until $D_{curr} = D_f$, and finally outputs a correct EP that the Kubernetes operator can follow to reach the desired system configuration.

Algorithm 1 FindEvolutionPlan

Require: D_0, D_f, M, $\text{CPU}_{\max}$, $EP \leftarrow [\,]$
Ensure: EP deployment plan from D_0 to D_f

$D_{\text{curr}} \leftarrow D_0$
$to\_add \leftarrow D_f \setminus D_{\text{curr}}$ ▷ Identifying required changes
$to\_remove \leftarrow D_{\text{curr}} \setminus D_f$
$removable \leftarrow \textsc{FindSafeToRemove}(to\_remove, D_{\text{curr}})$
for all $S_{j'}^{(i')}$ **do**
    $DS = DS \cup Remove(S_{j'}^{(i')})$
    $D_{\text{curr}} \leftarrow D_{\text{curr}} \setminus S_{j'}^{(i')}$
    $to\_remove \leftarrow to\_remove \setminus S_{j'}^{(i')}$
end for
deployable $\leftarrow \textsc{FindSafeToDeploy}(\text{to\_add}, D_{\text{curr}}, \text{CPU}_{\max})$
for all $S_{j'}^{(i')}$ in deployable **do**
    $\text{CPU}_{proj} \leftarrow \sum_{S \in D_{curr}} \text{cpu}(S) + \text{cpu}(S_j^{(i)})$
    if $\text{CPU}_{proj} \leq \text{CPU}_{\max}$ **then**
        $DS = DS \cup Deploy(S_{j'}^{(i')})$
        $D_{\text{curr}} \leftarrow D_{\text{curr}} \cup S_{j'}^{(i')}$
        $to\_add \leftarrow to\_add \setminus S_{j'}^{(i')}$
    else
        $removable \leftarrow \textsc{FindSafeToRemove}(to\_remove, D_{\text{curr}})$
        for all $S_{j'}^{(i')}$ **do**
            $DS = DS \cup Remove(S_{j'}^{(i')})$
            $D_{\text{curr}} \leftarrow D_{\text{curr}} \setminus S_{j'}^{(i')}$
            $to\_remove \leftarrow to\_remove \setminus S_{j'}^{(i')}$
        end for
    end if
end for
EP.append(DS)
if $D_{\text{curr}} = D_f$ **then return** EP
end if
$\textsc{FindEvolutionPlan}(D_0, D_f, M, \text{CPU}_{\max}, EP)$

FindSafeToRemove A microservice $S_j^{(i)} \in D_{curr}$ is considered safe to remove if it has no future or current dependencies, meaning that none of the interfaces it provides are required as external interfaces or consumed by any microservices in D_{curr} or D_f. Formally, this holds if

$S_{j'}^{(i')}) \cap I_p(S_j^{(i)}) = \emptyset$ and $I_p(S_j^{(i)}) \cap \mathcal{E}_{\text{req}} = \emptyset$ for all $S_{j'}^{(i')} \in to\_add \quad \cup D_{curr}$.

$S_j^{(i)} \in D_{curr}$ is also safe to be removed if, for each $v \in I_p(S_j^{(i)}) = \emptyset$, it must be provided by another microservice in the deployment or only consumed by microservices that are themselves redundant.

Suppose that the above conditions are not met *individually*. Then a set of microservices $R \subset D_{curr}$ might still be jointly removable if they form a dependency cycle and collectively satisfy the constraints, as shown by Algorithm 2.

Algorithm 2 FindSafeToRemove

Require: to_remove, D_{curr}
Ensure: Set of removable microservices
1: $removable \leftarrow \emptyset$
2: **for all** $S_j^{(i)} \in to\_remove$ **do**
3: **if** No $S_{j'}^{(i')}$ in $D_{curr} \cup D_f$ depends on any $v \in I_p(S_j^{(i)})$ **and** $v \notin \mathcal{E}_{req}$ **then**
4: $removable \leftarrow removable \cup \{S_j^{(i)}\}$
5: **else if** All $v \in I_p(S_j^{(i)})$ are either provided by others or only used by redundant services **then**
6: $removable \leftarrow removable \cup \{S_j^{(i)}\}$
7: **end if**
8: **end forreturn** $removable$

FindSafeToDeploy A microservice $S_j^{(i)} \in D_{curr}$ is considered safe to deploy if, for each $v \in I_c(S_j^{(i)})$, it must be provided by another microservice in D_{curr}, and the system remains within the CPU quota CPU_{max}. Similarly to finding safe microservices to remove, there might exist cycles of microservices that can be deployed jointly, as described by Algorithm 3.

5 Experimental evaluation

The experimental setup was based on a conceptual architecture that splits an existing multi-version RAN application into 14 microservices, each available in different versions, collectively supporting 48 interfaces designed by Ericsson AB for Cloud Deployment and managed by Kubernetes. A local Kubernetes cluster served as the test environment for the evaluation, with each microservice represented as a Kubernetes Deployment and interfaces as Kubernetes Services. A Custom Resource Definition (CRD) of the kind deployment update was established and outlined in a manifest file, specifying the universe of deployable

Algorithm 3 FindSafeToDeploy

Require: to_add, D_{curr}, $\text{CPU}_{\max}$
Ensure: Set of deployable microservices
1: $deployable \leftarrow \emptyset$
2: **for all** $S_j^{(i)} \in to\_add$ **do**
3: **if** All $v \in I_c(S_j^{(i)})$ are provided by some $S_{j'}^{(i')} \in D_{curr}$ **then**
4: $\text{CPU}_{proj} \leftarrow \sum_{S \in D_{curr}} \text{cpu}(S) + \text{cpu}(S_j^{(i)})$
5: $deployable \leftarrow deployable \cup \{S_j^{(i)}\}$
6: **end if**
7: **end forreturn** $deployable$

microservices M, their supported interface versions, and the external interface requirements $\mathcal{E}_{req}$. The CRD enabled the implementation of the operator pattern, a software extension to Kubernetes that listens to custom resource modifications and updates the environment. The data provided includes the amount of resources (CPU and memory) that every microservice consumes, measured beforehand in internal laboratories. Based on that, suitable requests and limits are set for each microservice to use at runtime. These values are always specified in advance and not adjusted during run-time. Kubernetes uses these requests and limits to allocate resources to each container accordingly.

5.1 Generation of problem instances

Interface samples for problem instances were selected based on historical system data, which contained a total of 4261 time-stamped interface updates over eight years. Problem instances are created based on the assumption that updates to the external interfaces typically drive a system upgrade. The samples were generated by organizing the data into weekly and daily samples, capturing each update to the external interface. A total of 67 interface samples were generated: 51 weekly samples from 2022, as this year represents the highest update frequency with 40 (out of 48) distinct interface updates; 16 daily samples from February 2024, as this month contains the most samples with external interface updates. Furthermore, the microservices supporting the newly upgraded interfaces were sampled in three ways, with varying levels of complexity, as follows.

- **Large:** This sample presents the largest and most complex search space, where all providers and consumers of interfaces that are subject to an upgrade are upgraded to support the new version. This implies that the minimal solution typically involves one version of each microservice,
- **Medium:** All providers are upgraded. Among consumers, half are randomly selected to be upgraded to support the new version.
- **Small:** All providers are upgraded. For each updated interface, exactly one consumer is randomly selected to be upgraded. If a microservice consumes multiple updated interfaces, it is upgraded to support only the selected one.

Our intuition is that while medium and smaller problem instances are easier to solve due to the smaller search space, their final solution will likely be less resource-efficient. Since not all microservices are upgraded, multiple versions usually need to coexist in the final deployment to satisfy dependencies, resulting in a solution that consumes more CPU than a solution to the large sample.

Each sample $s \in \mathcal{S}$ thus generates three problem instances, giving a total of $3 \cdot 67 = 201$ instances. Each problem instance serves as an input to the algorithm, where the initial deployment state D_0 consists of one version of each microservice, the set of required interfaces $\mathcal{E}_{req}$ is extended to include the updated interfaces in the corresponding interface sample. For each sample, the universe of deployable microservices M is then extended to include the microservice versions selected by the sampling scheme.

The interface sample that constitutes the Large problem instances is available for sharing with the research community and can be found in Appendix C, Table C.1 of the author's previous work, which forms the basis of this paper [15].

5.2 Baselines and evaluation metrics

In addition to DEP-DS, two baseline algorithms, Ideal Dependency Resolver and Scheduler (IDRS), and Dependency Resolver and Scheduler with Stratified Pruning (DRS-SP), were implemented for the evaluation:

1. Ideal Dependency Resolver and Scheduler (IDRS): A tree-based algorithm to find all EPs from D_0 to any valid deployment state D_k, which then chooses the EP that minimizes the total system resource usage (CPU request).
2. Dependency Resolver and Scheduler with Stratified Pruning (DRS-SP): A modification of IDRS that utilizes a stratified sampling technique to sample child nodes based on their size. Each node represents one or more deployment stage, and the size is denoted by the number of microservices to be deployed in that stage. If the size exceeds 30, this approach partitions and chooses the 10 largest, 10 in the middle, and 10 smallest child nodes.

The following metrics were used for evaluation and calculated for all three samples in each algorithm setting:

1. Success Rate: The ratio of the successfully solved problem instances.
2. Resource Efficiency (CPU_{avg}): The average CPU request, that is, the sum of the CPU requests for microservices in each of the problem instances that the algorithm successfully solved, divided by the number of solved ones.
3. Average Response Time (T_{avg}): Average time taken to produce an evolution plan (for the problem instances that the algorithm successfully solved).

5.3 Evaluation outomes

Table 1 shows a comparative evaluation of the three algorithms with the data created using the three sampling strategies (Large, Medium, Small), and in total for the three samples.

Table 1: Evaluation outcomes

	Large	**Medium**	**Small**	**Total**
Success Rate				
IDRS	84%	95.6 %	97%	92.5%
DRS-SP	97%	97%	97%	97.0%
DEP-DS	**100**%	**100**%	**100**%	**100**%
T_{avg}**(s)**				
IDRS	15.10	11.90	0.35	9.11
DRS-SP	0.92	0.95	1.15	1.00
DEP-DS	2.46	0.69	0.2	1.15
CPU_{avg}**(mCPU)**				
IDRS	480.80	530.00	532.58	512.20
DRS-SP	480.69	533.77	532.58	515.59
DEP-DS	480.00	530.23	534.23	514.47

Based on the success rate, T_{avg} and CPU_{avg}, the main findings are that while the algorithms have similar performance in terms of CPU_{avg}, DEP-DS is the only algorithm capable of resolving all problem instances (success rate 100% for all samples), and with a shorter response time compared to IDRS and DRS-SP.

Overall, as seen in the CPU_{avg} section of the table, IDRS can explore larger search spaces and identify solutions with lower CPU consumption in smaller problem instances. However, it faces scalability issues as problem complexity increases, as evidenced by a decreased success rate for instances in the Large sample.

In addition, we tried to see if DEP-DS could be made even faster by adopting a greedy resolution instead of branch and bound. It turned out that it would be faster to resolve all three instances, but at the cost of a lower success rate (86%, 95%, and 86% for Large, Medium, and Small, respectively) compared to the third row in the table (100% for all instances).

Moreover, DEP-DS running with greedy search gave a CPU_{avg} that grew substantially for all three samples (624.52, 613.57, and 615.56 for Large, Medium, and Small, respectively). This suggests that a greedy approach is ideal when fast evolution plan generation is prioritized, although it may come with higher resource usage. Given that our objective was to minimize the CPU usage, DEP-DS outperforms the greedy approach given that the response time, row 6 of Table 1 is acceptable even for the large sample.

6 Conclusions and Future works

This research shows that DEP-DS is a promising method for solving the Microservice Dependency Problem and the Evolution Planning Problem, evidenced by its perfect success rate and good resource efficiency during evaluation. DEP-DS effectively balances solution quality, making it more suitable for larger, more intricate instances.

When solving MDP and EPP, our first priority was to find something workable, leaving room for improvement when it comes to optimizing the branch and bound algorithm (through a lower bound calculation and heuristic selection), and even other heuristics to replace branch and bound.

We found that obtaining relevant data to base the evaluation on was in itself a major challenge. To obtain historical data for prototypes evaluated by researchers is not an easy task, and we were able to solicit authentic data to illustrate that the solutions would be workable in a real context. The samples that were selected according to the criteria in Section 5 are shared with the research community, and we welcome new approaches that use the same data and novel algorithms to further this work.

The problem instances were generated from calendar days with external interface updates and selected based on predicted difficulty. They are therefore to be seen as an approximation to the worst-case scenarios based on historical data. Studying scenarios based on other samples that would be more representative of an average day would be another problem to look at. This could be done, for example, by choosing days at random, which would be another direction for future work. Further studies on real-world conditions, such as failures, latency, and dynamic resource changes are needed.

Acknowledgements

The Second author was supported by ELLIIT, Excellence Center at Linköping-Lund on Information Technology. The authors wish to thank Erik Malmberg, Klas Strömberg, and Elisabeth Sjöstrand at Ericsson AB for discussions on our work and for providing the historical data for experimental evaluations. The authors also wish to thank Soheil Samii at the Department of Computer Science (IDA) for supporting the research.

References

1. Merkouche, S., Haroun, T., Bouanaka, C., Smaali, M.: TERA-Scheduler for a Dependency-based Orchestration of Microservices. In: International Conference on Advanced Aspects of Software Engineering (ICAASE), pp. 1–8. IEEE (2022). https://doi.org/10.1109/ICAASE56196.2022.9931568
2. He, X., Tu, Z., Liu, L., Xu, X., Wang, Z.: Optimal evolution planning and execution for multi-version coexisting microservice systems. In: Service-Oriented Computing: 18th International Conference, ICSOC, pp. 3–18. Springer (2020). https://doi.org/10.1007/978-3-030-65310-1_1
3. Akhter, F.: A heuristic approach for minimum set cover problem. In: International Journal of Advanced Research in Artificial Intelligence, pp. 40–45. Citeseer (2015). https://doi.org/10.14569/IJARAI.2015.040607
4. Coudert, O.: On solving covering problems [logic synthesis]. In: 33rd Design Automation Conference Proceedings, pp. 197–202. IEEE (1996). https://doi.org/10.1109/DAC.1996.545572
5. Le Berre, D., Parrain, A.: On SAT technologies for dependency management and beyond. First Workshop on Software Product Lines (ASPL'08) **2**, 197–200 (2008)
6. Edelkamp, S., Schrödl, S.: Heuristic search: theory and applications. Academic Press (2012), ISBN: 978-0-12-372512-7
7. Florisson, M., Mycroft, A.: Towards a Theory of Packages. ACM SIGPLAN Notices (2017). https://doi.org/10.1145/800225.806833
8. Gholami, S., Goli, A., Bezemer, C., Khazaei, H: A Framework for Satisfying the Performance Requirements of Containerized Software Systems Through Multi-Versioning. ICPE'20 (2020). https://doi.org/10.1145/3358960.3379125
9. Nilsson, U., Maluszynski, J.: LOGIC, PROGRAMMING AND PROLOG (2ED). John Wiley and Sons Ltd (1995), ISBN: 9780471959960
10. Chait-Roth, D., Namjoshi, K.S., Wies, T.: Consistent Updates for Scalable Microservices. CoRR abs/2508.04829 (2025). https://doi.org/10.48550/ARXIV.2508.04829
11. Zhang, Y., Yang, J., Jin, Z., Sethi, U., Rodrigues, K., Lu, S., Yuan, D.: Understanding and Detecting Software Upgrade Failures in Distributed Systems. SOSP'21: ACM SIGOPS 28th Symposium on Operating Systems Principles (2021), pp. 116–131. ACM. https://doi.org/10.1145/3477132.3483577
12. Shi, Q., Xie, X., Fu, X., Di, P., Li, H., Zhou, A., Fan, G.: Datalog-Based Language-Agnostic Change Impact Analysis for Microservices. 47th IEEE/ACM International Conference on Software Engineering (ICSE) (2025), pp. 78–89. IEEE. https://doi.org/10.1109/ICSE55347.2025.00115
13. Pham, M.C., Truc, M.T., Hoang, X.T., Nguyen, K-K.: Energy-Efficient Update of Microservices Applications in Kubernetes Clusters. 16th International Conference on Knowledge and System Engineering (KSE) (2024), pp. 167–172. IEEE. https://doi.org/10.1109/KSE63888.2024.11063630
14. Wang, Y., Conan, D., Chabridon, S., Bojnourdi, K., Ma, J.: Runtime models and evolution graphs for the version management of microservice architectures. 28th Asia-Pacific Software Engineering Conference (APSEC) (2021). IEEE. https://doi.org/10.1109/APSEC53868.2021.00064
15. Witt, E.: Dependency-Based Orchestration of a Multi-Version Microservice RAN Application. Master's thesis, Linköping University (2024), https://liu.diva-portal.org/smash/record.jsf?pid=diva2%3A1906848&dswid=8369