YUMA – An AI Planning Agent for composing IT Services from Infrastructure-as-Code Specifications

Florian Baer
Accenture
Philipps University of Marburg
baer.florian@gmail.com

Michael Leyer
Philipps University of Marburg
Queensland University of Technology
michael.leyer@wiwi.uni-marburg.de

Abstract

Infrastructure-as-code enables cloud architects to automate IT service delivery by specifying IT services through machine-readable definition files. To allow for a reusability of the infrastructure-as-code specifications, cloud architects specify IT services as compositions of sub-processes. As the AI planning agents for automated IT service composition proposed by prior research fall short in the infrastructure-as-code context, we design a search-based problem-solving agent named YUMA according to a design science research process to fill this research gap. YUMA holds a search tree reflecting the state space and transition model. It includes an algorithm for building the search tree and two algorithms for determining the minimum composition plan. The underlying IT service composition problem is explicated for the infrastructure-as-code context and formulated as a search problem. The results of the demonstration and evaluation show that YUMA fulfills the requirements necessary to solve this problem and digitizes an important task of cloud architects.

Keywords: IT service, Service composition, Infrastructure-as-code, Artificial intelligence.

1. Introduction

Cloud infrastructures are becoming more and more popular for enterprises, but the diversity of IT services that can be delivered to cloud environments requires automated yet individual compositions. In this regard, infrastructure-as-code (IaC) is promoted referring to a paradigm that argues for the specification of IT services, which must be delivered to manage IT infrastructures, through machine-readable definition files (i.e., code) (Sandobalín et al., 2020). By specifying IT services according to IaC, the delivery of IT services to cloud environments can be automated (Chiari et al., 2022). Hence, cloud architects use IaC tools, such as Ansible, Puppet, and Chef, to compose IT services from imperative IaC specifications (Kumara et al., 2021).

The current state has however some inefficiencies that hinder enterprises to fully exploit the potential of IT service composition. Cloud architects must know which sub-processes have to be executed in which order to compose an IT service. However, when specifying sub-processes using IaC tools, cloud architects must specify and update the dependencies between them manually. In enterprises, the sub-processes are stored in repositories that are updated regularly, i.e., new sub-process specifications are added and existing ones are altered. To compose an IT service, cloud architects must first scan the repositories to discover relevant sub-processes and examine their IaC specifications to determine valid execution orders. This makes the manual composition of IT services from IaC specifications a time-consuming activity for cloud architects. It ranges from a few minutes to several hours depending on the number of sub-processes maintained in the enterprise, number of repositories the sub-processes are stored in, complexity of the IaC specifications, and expertise of the cloud architect in reading and writing IaC specifications. The higher this effort is expected by cloud architects, the higher is the risk of cloud architects re-specifying already specified sub-processes. However, this would be a violation of IaC best practices (Kumara et al., 2021), as it would result in redundant work, which can sum up to several days or even a few weeks of redundant effort in an enterprise (reported by anecdotal evidence since no empirical studies are available).

A commonly accepted way to overcome these inefficiencies is the automation of the IT service composition (Rao & Su, 2005). Prior research has proposed different artificial intelligence (AI) planning agents for automated IT service composition (Jula et al., 2014). However, in the IaC context, these agents have two major shortcomings. First, the agents solve IT service composition problems formulated specifically for the domains of automated web and cloud service composition. Hence, the problem formulations consider the specific characteristics of these domains (Jula et al., 2014; Zou et al., 2014), but not those specific to the IaC domain. Second, because of these problem definitions,

URI: https://hdl.handle.net/10125/102759
978-0-9981331-6-4

these agents are designed in a way so that they interact with known environments, but not unknown environments as required in the IaC context (Hatzi et al., 2015; Kuzu & Cicekli, 2012). To close this research gap, we define the following research question: *What is the design of an AI planning agent supporting cloud architects at automatically composing IT services based on IaC specifications?*

We address this research question by designing a search-based problem-solving agent named (y)et another (u)nique (m)achine (a)gent (YUMA). The design knowledge that is contributed to the λ knowledge base must be categorized as invention and exaptation (Gregor & Hevner, 2013). We are first in formulating the IT service composition problem for the IaC context. We also draw from algorithm design and analysis techniques and apply them to the design of YUMA. The requirements agents must fulfill to solve the IT service composition problem in the IaC context and the algorithms implemented by YUMA in pseudocode are described. Next to these level 2 artifacts, we contribute to research an instantiation of YUMA.

The article is organized as follows: In Section 2 we provide an overview on the theoretical background including the conceptual aspects with regard to IT services and AI planning agents as well as related work. Our research method is then described in Section 3. Following the research method, Section 4 provides the problem explication while the requirements are presented in Section 5. We then describe the design of YUMA in Section 6 with a demonstration in Section 7. The evaluation is presented in Section 8. We discuss our approach and conclude the article in Section 9.

2. Theoretical Background

2.1. IT Services

This study is rooted in service operations research (Sampson, 2012; Sampson & Froehle, 2006). Accordingly, we define a service as a "[…] type of process, and 'services' are multiple service processes" (Sampson, 2012, p. 183). A service represents a sequence of actions, whose execution allows the production of a desired outcome (Yalley & Sekhon, 2014). In line with this definition, a service can be formalized as a directed graph (Becker et al., 2009).

We can describe a service by a vector of attributes $S = (A, P, F)$ (Baer & Leyer, 2016). The set of actions, which are included in the sub-processes of the service, is denoted as $A = \{a_1, a_2, \dots, a_n\}$, with $n \in \mathbb{N}$. The sub-processes, which are included in the service, are defined by the set $P = \{p_1, p_2, \dots, p_m\}$, with $m \in \mathbb{N}$. Each p_i, $\forall i \in \mathbb{N}$, $1 \leq i \leq m$, is a subset of A: $p_i \subset A$, with $p_1 \cup p_2 \cup \dots \cup p_m = A$. The control-flow $f_i \subseteq p_i \times p_i$, $f_i \in F = \{f_1, f_2, \dots, f_m\}$ of a sub-process p_i describes the execution order of actions included in the sub-process p_i through constructors permitting flow of execution control (e.g., sequence, condition, and parallelism) (van der Aalst et al., 2003). Each element $(a_j, a_k) \in f_i$ is a constructor ω_{a_j,a_k}, $\forall j, k \in \mathbb{N}, \forall a_j, a_k \in p_i$, which represents a propositional formula. If $\omega_{a_j,a_k} = 1$, the execution of action a_k follows the execution of a_j.

Exemplary IT services composed by cloud architects are platform services, such as Kubernetes. For instance, a single node Kubernetes cluster can be specified as a composition of a set of sub-processes setting up the required virtual machines (VMs), control plane and node components, container runtime (e.g., Docker), and a container network fabric (e.g., Flannel).

2.2. AI Planning Agents

We draw from the notion that AI is the study of rational agents (Russell & Norvig, 2016; Sutton & Barto, 2018). An agent is an entity, which perceives its environment and acts upon that environment. It will be rational, if it performs only those actions, which are expected to support it achieving its goals, based on its perceptions of the environment and its knowledge about the environment. Therefore, an important part of AI is planning, i.e., "[…] devising a plan of action to achieve one's goals […]" (Russell & Norvig, 2016, p. 366). There are different types of AI planning agents, such as search-based problem-solving agents and hybrid propositional logical agents, depending on the formulation of the problem to be solved.

An agent can be characterized along three dimensions according to the model-inference-learning paradigm (Liang & Sadigh, 2019):

- **Model:** A formal description of the environment, with which the agent interacts. It can be an accurate or approximate representation of (parts of) the environment. It reflects the agent's knowledge about the environment.
- **Learning:** Adjustment (i.e., update of the parameters of the model) of an incomplete model over time based on feedback perceived from the environment.
- **Inference:** Solving a problem based on one or more algorithms with respect to the model.

The environment, with which an agent interacts, can be characterized along a set of continuums (Russell & Norvig, 2016). Exemplary continuums are described in Table 1.

Table 1. Properties of agent environments.

Fully observable	**Unobservable**
The agent perceives the complete state of the environment at any time.	The agent does not perceive the environment at all.
Deterministic	**Stochastic**
The successor state is only determined by the current state and performed action.	An action performed at a specific state can result in different successor states with different probabilities.
Episodic	**Continuing**
An interaction between the agent and environment ends in a terminal state, which is followed by a reset of the environment.	The interaction between the agent and environment goes on infinitely.
Static	**Dynamic**
The environment cannot change during the interactions with the agent.	The environment can change during the interactions with the agent.
Known	**Unknown**
The agent has a complete and accurate model of the environment.	The agent must first explore the environment and build a model of it.

2.3. Related Work

AI planning has been well studied in the fields of automated web and cloud service composition (Jula et al., 2014; Rao & Su, 2005). There exist several literature reviews about these topics and well known and commonly integrated AI planners (Masdari et al., 2021; Razian et al., 2022). Table 2 summarizes the AI planners integrated by most of the related AI planning agents.

Table 2. Summary of related AI planning agents.

Agent	**Characteristics**
SHOP2	**Model:** Hierarchical task network (HTN) **Learning:** Knowledge base containing operators and methods **Inference:** SHOP2 search algorithm
OWLS-XPlan	**Model:** Connectivity graph and relaxed planning graph **Learning:** *BuildRelaxedPlanningGraph* **Inference:** Enforced hill-climbing
(Kuzu & Cicekli, 2012) (Simplanner)	**Model:** Search tree and Relaxed planning graph **Learning:** Graphplan extension and action selection process **Inference:** (Real-time) Depth-first search with backjumping
(Zou et al., 2014) (Metric-FF and SATPlan)	**Model:** Relaxed planning graph and conjunctive normal form (CNF) sentences **Learning:** Relaxed Graphplan and Knowledge base (i.e., a set of CNF sentences) **Inference:** Enforced hill-climbing and SATPlan

The related AI planning agents are designed in a way so that they require a repository of composite IT services described by ontologies (e.g., web ontology language for web services (OWL-S) or web services description language (WSDL)) or tree-based structures (Eshuis & Mehandjiev, n.d.; Hatzi et al., 2015). The composition requests are performed against these repositories. Due to this design, the agents require users to know about and specify the dependencies (i.e., preconditions) of the relevant processes as part of the composite service specifications. This works for the agents, because they divide users into service providers and requesters (Kuzu & Cicekli, 2012; Zou et al., 2014). Only the service providers contribute specifications of composite IT services to the repositories. Therefore, the agents interact with known environments and do not have to deal with model learning. However, in the IaC context, there cannot be made such a differentiation of users. Each cloud architect represents a sub-process contributor while also being a service requester. Cloud architects understand the dependencies between those sub-processes specified by themselves. But, they do not know about the dependencies of sub-processes specified by others before exploring and analyzing the related IaC specifications. That is why, in the IaC context, AI planning agents must interact with unknown environments. With YUMA, we close this research gap.

3. Research Method

To address our research question, we design a search-based problem-solving agent named YUMA. As the agent represents an artifact, we apply a design science research (DSR) approach to its design. We follow the method framework for DSR (Johannesson & Perjons, 2014), as it is well accepted in the information systems engineering field (Jouck & Depaire, 2018). Our research process is shown in Figure 1.

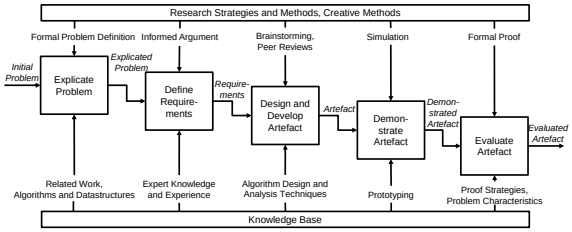

Figure 1. The adopted research process.

To explicate the problem, we define the IT service composition problem for the IaC context. For this research, we formulate the IT service composition problem as a search problem.

As input from the relevance cycle, we define a set of requirements, which must be fulfilled by any agent tackling the formulated IT service composition

problem. These requirements are defined based on informed arguments drawing from the authors' expertise and experience in the application domain.

YUMA implements three algorithms to fulfill the defined requirements. To come up with these algorithms, we applied algorithm design and analysis techniques drawn from the rigor cycle. A backtracking algorithm is performed to build the required search tree based on a given set of sub-processes. In the first iteration of the design and rigor cycle, YUMA applied a dynamic programming (DP) algorithm to determine the required sub-processes and their execution order for composing a specific IT service. However, because of its quadratic worst-case running time, in the second design and rigor iteration, the DP algorithm is replaced by a Uniform Cost Search (UCS). A formatting algorithm is applied to derive the execution order of the sub-processes from the UCS results.

YUMA is implemented in the programming language Go to demonstrate its feasibility.[1] As a first output to the relevance cycle, the YUMA instantiation is simulated on two example cases inspired by real-world cases experienced by one of the authors during his work as cloud architect. First, the installation of WordPress and phpBB on an Amazon Web Services (AWS) Elastic Compute Cloud (EC2) instance. Second, the setup of a single node Kubernetes cluster on an AWS EC2 instance. For both example cases, distinct sets of Ansible roles were developed by the authors.[2]

According to the framework for evaluation in DSR (FEDS), our evaluation represents an ex post evaluation in an artificial setting (Venable et al., 2012). We evaluate YUMA by proving the correctness of the algorithms applied by it. The correctness of the backtracking algorithm is proven by induction. The correctness of the formatting algorithm is proven by a loop invariant.

4. Problem Explication

4.1. IT Service Composition Problem

The IT service composition problem for the IaC context can be defined as the following. Given a set of sub-processes $P = \{p_1, p_2, \dots, p_m\}$, the set of actions $A = \{a_1, a_2, \dots, a_n\}$ included in the sub-processes, the corresponding control-flows $f_i \subseteq p_i \times p_i,\ f_i \in F = \{f_1, f_2, \dots, f_m\}$, $\forall m, n, i \in \mathbb{N},\ 1 \leq i \leq m$, and a specific $p^* \in P$, determine the minimal execution order (MEO) of $S^* = (A^*, P^*, F^*)$.

To determine the MEO of S^*, a total order $\rightarrow$ on P must be determined first. The characteristics of this total order are described in Table 3.

Let $\mathcal{P}_{p^*}(P) \subseteq \mathcal{P}(P)$ be the set of all subsets of P, in which p^* is the last executable sub-process, i.e. $p_i \rightarrow p^*$, with $p^* \in \mathrm{P}',\ \forall p_i \in P',\ \forall P' \in \mathcal{P}_{p^*}(P)$. Furthermore, we define $P^* = \underset{P' \in \mathcal{P}_{p^*}(P)}{\arg\min} |P'|$, with ties broken arbitrarily, as the minimal execution set (MES) including only those sub-processes, which must be executed before p^* can be executed. Let $A^* \subseteq A$ be the set of actions, which are included in the sub-processes included in P^* and let $F^* \subseteq F$ be the set of control-flows of the sub-processes included in P^*. Then, $\rightarrow$ is a total order on P^*, the MES, and therefore it is the MEO (of S^*) to be determined.

Table 3. Characteristics of a total order on *P*.

Characteristic	Description
$p_i \rightarrow p_i, \forall p_i \in P$	$\rightarrow$ is reflexive, i.e., once p_i has ben executed, it can be executed again in the future
If $p_k \rightarrow p_j$ and $p_j \rightarrow p_i$, then $p_k \rightarrow p_i$, $\forall p_i, p_j, p_k \in P$	$\rightarrow$ is transitive, i.e., if p_j can only be executed after p_k has been executed and p_i can only be executed after p_j has been executed, then p_i also can only be executed after p_k has been executed
If $p_i \rightarrow p_j$ and $p_j \rightarrow p_i$, then $p_i = p_j$, $\forall p_i, p_j \in P$	$\rightarrow$ is antisymmetric, i.e., if p_i can only be executed after p_j has been executed and p_j can only be executed after p_i has been executed, then both p_i and p_j are the same sub-process
Either $p_i \rightarrow p_j$ or $p_j \rightarrow p_i, \forall p_i, p_j \in P$	$\rightarrow$ is a total order on P, i.e., any pair of sub-processes is comparable regarding the execution order

While p^* represents the composition request, the MEO represents the minimum composition plan and therefore the optimal composition solution to the defined IT service composition problem (Zou et al., 2010, 2014).

4.2. Search Problem

For this research, we formulate the IT service composition problem as a search problem. Therefore, we define $c_i \in C = \{c_1, c_2, \dots, c_m\},\ \forall p_i \in P$, as $c_i = 2^i - 1$ being the binary representation of p_i. Let $c^* \triangleq p^*$ and $C^* \triangleq P^*$ be the binary representation of p^* and the MES, respectively.

C enables the representation of all possible states $s_r \in \mathcal{S}_S = \{s_0, s_1, \dots, s_{2^m-1}\},\ \forall r \in \mathbb{N},\ 0 \leq r \leq 2^m - 1$

[1] The source code of YUMA can be found on GitHub: https://github.com/Floble/go-utils/blob/ucs/algorithms/artificialintelligence/search/yuma.go.

[2] The Ansible roles can be found on GitHub: https://github.com/Floble/ansible-utils.

resulting from the execution of any sub-process $p_i \in P$. Because each sub-process $p_i \in P$ has a binary representation $c_i \in C$, each $s_r \in \mathcal{S}_S$ is also represented as a binary value. The execution of sub-processes $p_i \in P$ at specific states $s_r \in \mathcal{S}_S$ can be described in the form of state-sub-process pairs (s_r, c_i), $\forall s_r \in \mathcal{S}_S, c_i \in C$. Therefore, the successor states resulting from (s_r, c_i)-pairs can be determined by applying logical (Boolean) operators to each (s_r, c_i)-pair. We define the functions $or: \mathcal{S}_S, C \to \mathbb{N}^+$, $and: \mathcal{S}_S, C \to \mathbb{N}^+$, and $xand: \mathcal{S}_S, C \to \mathbb{N}^+$ to describe the application of the logical operators OR, AND, and XAND on specific (s_r, c_i)-pairs, respectively. In general, following the MEO of S^* will result in a sequence of (s_r, c_i)-pairs, which represents the shortest path from the initial state (i.e., s_{start}) of the search tree to a target state, i.e., a state at which p^* has been executed. This shortest path is denoted as $Path^* = ((s_r, c_i)_d)_{d=0}^{d^*}$, with $d^* \in \mathbb{N}^+$, $0 \leq d^* \leq m$.

The IT service composition problem is formulated as a search problem as described in Table 4 (Russell & Norvig, 2016).

Table 4. Formulation of the search problem.

Component	Description
States: $s_r \in \mathcal{S}_S = \{s_0, s_1, \dots, s_{2^m-1}\}, \forall r \in \mathbb{N}, 0 \leq r \leq 2^m - 1$	A state s_r reflects the state of the environment, on which all or a subset of the sub-processes $p_i \in P$ have been executed. Each $s_r \in \mathcal{S}_S$ conveys all required information to make decisions about which sub-process $p_i \in P$ to execute next without the need to consider how the current state was reached.
Initial state: $s_{start} \in \mathcal{S}_S$ and $s_{start} = 0$ **End state:** $s_{end} \in \mathcal{S}_S$ and $s_{end} = 2^m - 1$	The initial state is always 0, i.e., none of the sub-processes $p_i \in P$ has been executed from the start. The ending state is always $2^m - 1$, i.e., all sub-processes $p_i \in P$ has been executed in the end.
Actions: $\mathcal{A}(s_r) = \cup_{c_i \in C} action(s_r, c_i), \forall s_r \in \mathcal{S}_S$ $action(s_r, c_i) = \begin{cases} \{\}, & and(s_r, c_i) \neq 0 \\ \{c_i\}, & \text{otherwise} \end{cases}$	All the sub-processes $p_i \in P$, which have not been executed yet, can be executed at state $s_r \in \mathcal{S}_S$.
Transition model: Let $\mathcal{M}$ be a $2^m - 1 \times m$ matrix, which represents the model (i.e., the search tree) of YUMA, $\mathcal{M}_{r,i}$ represents the successor state reached when executing c_i at s_r. $successor(s_r, c_i) = \mathcal{M}_{r,i} = \begin{cases} or(s_r, c_i), & \text{if } p_i \text{ can be executed at } s_r \\ s_r, & \text{otherwise} \end{cases}$, $\forall s_r \in \mathcal{S}_S, c_i \in C$	Whether or not p_i can be executed at s_r must be explored when interacting with the environment, i.e., p_i is performed at s_r and the result of this execution must be observed. This result is represented in the search tree $\mathcal{M}$ that must be built. Hence, any successor state can be determined by a lookup in $\mathcal{M}$.
Goal test: $isEnd(s_r) = \begin{cases} true, & \text{if } s_r = s_{end} \\ false, & \text{otherwise} \end{cases}$ $isTarget(s_r, c^*) = \begin{cases} true, & if\ and(s_r, c^*) \neq 0 \\ false, & \text{otherwise} \end{cases}$	If all sub-processes $p_i \in P$ have been executed, the end state will be $2^m - 1$ representing a state, at which all sub-processes $p_i \in P$ have been executed. A state s_r will be a target state, if it holds that p^* already has been executed.
Path cost: $cost(s_r, c_i) = 1, \forall s_r \in \mathcal{S}_S, c_i \in C$	The cost of executing a sub-process $p_i \in P$ at state $s_r \in \mathcal{S}_S$ is always 1.

5. Requirements

Cloud architects interact with cloud environments (e.g., AWS and Microsoft Azure). In the light of the circumstances that cloud architects must deal with when composing IT services based on IaC specifications (see section 1), cloud environments must be characterized as described in Table 5.

Table 5. Cloud environments in the IaC context.

ID	Property	Explanation
P1	Fully observable	IaC tools, such as Ansible, can gather facts about the current state of the cloud environment (e.g., an AWS EC2 instance).
P2	Deter-ministic	The execution of an IaC-based sub-process is either successful (i.e., the result is a new state) or unsuccessful (i.e., no state change).
P3	Dynamic	IaC-based sub-processes are stored in repositories that are regularly updated (i.e., sub-processes are added and altered).
P4	Unknown	In large enterprises, many cloud architects specify IaC-based sub-processes. A single cloud architect does not know all the dependencies between the sub-processes but must first explore or analyze them.

To solve the search problem formulated in section 4.2., a search-based problem-solving agent must interact with such cloud environments and therefore must fulfill the requirements, which are listed in Table 6, along its three dimensions.

Table 6. Requirements for an AI planning agent.

ID	Dimension	Description
R1	Model	Because of P1 and P2, the agent must hold a search tree reflecting the state space and the transition model (i.e., the dependencies between the sub-processes).
R2	Learning	Because of P3 and P4, the agent must regularly explore the transition model by executing the IaC-based sub-processes in the cloud environment and build the search tree accordingly. The interactions with the cloud environment can be episodic and continuing.
R3	Inference	Because of R1, the agent must identify the shortest path from the initial state to a target state in the search tree in acceptable time.

6. Design

According to the model-inference-learning paradigm, the design of YUMA can be characterized within the following dimensions:

- **Model:** Search tree (see the description of the transition model in Table 4).
- **Learning:** BuildSearchTree (see section 6.1.)
- **Inference:** DetermineExecutionOrder (see section 6.2.); FormatExecutionOrder (see section 6.3.)

YUMA holds a search tree that is built by episodic interactions with the cloud environment. With the episodic interactions, YUMA performs the algorithm *BuildSearchTree*. The MEO is determined as the shortest path from the initial state to a target state in the built search tree. Towards that end, YUMA performs the algorithms *DetermineExecutionOrder* (i.e., an UCS) and *FormatExecutionOrder*.

6.1. Build Search Tree Algorithm

The algorithm *BuildSearchTree* is described by Table 7. We define $\sigma \in \mathbb{N}^{+}_{\setminus\{0\}}$ as the error acceptance rate. It is included in *BuildSearchTree*, because sometimes although the execution of a sub-process p_i at a state s_r should be successful, the execution still fails due to some temporary technical issues (e.g., a short outage of the internet connection). To overcome this issue, we recommend defining $\sigma > 1$. The parameter $d \in \mathbb{N}^{+}$ represents the current depth in the search tree. *BuildSearchTree* is initiated with $d = 0$, i.e., the depth of the initial state.

To simplify the time complexity analysis, we define the lines 5, 6, 7, 8, 10, 11, 13, and 17 to take constant time. If $Path$ is implemented as a stack, lines 19 and 21 will represent the push and pop operations, respectively. Both operations take $O(1)$ time.

Table 7. The backtracking algorithm.

BuildSearchTree(σ, s_r, d, $Path$)

Parameters:
$\sigma \in \mathbb{N}^{+}_{\setminus\{0\}}$, $s_r \in \mathcal{S}_S$, $d \in \mathbb{N}^{+}$, $Path: \mathbb{N}^{+}_{\leq m} \rightarrow \mathcal{S}_S \times C$
Initialization:
$\mathcal{M}_{r,i} = 0, \forall r, i \in \mathbb{N}, 1 \leq r \leq 2^m - 1, 1 \leq i \leq m$
$Path = ()$
Algorithm:

```
1   if isEnd(s_r)
2       return M
3   for each c_i ∈ A(s_r)
4       for l = 1 to σ
5           create new VM in the cloud environment
6           execute each (s_r, c_i) ∈ Path on the VM
7           if ∃(s_r, c_i) ∈ Path(execution of (s_r, c_i) failed)
8               delete the VM in the cloud environment
9               continue
10          execute c_i on the VM
11          if execution of c_i is successful
12              M_{r,i} = or(s_r, c_i)
13              delete the VM in the cloud environment
14              break
15          else
16              M_{r,i} = s_r
17              delete the VM in the cloud environment
18      if successor(s_r, c_i) is equal to or(s_r, c_i)
19          Path = Path ⊎_asc (s_r, c_i)
20          BuildSearchTree(σ, successor(s_r, c_i), d + 1, Path)
21          Path = Path\(s_r, c_i)_d
22  return M
```

The base case, when $m = 0$, takes constant time: $T(0) = \Theta(1)$. The recursive case, when $m > 0$, takes the following time: $T(m) = mT(m-1) + \Theta(m)$. Thus, the asymptotic tight bound of the worst-case running time of *BuildSearchTree* is $\Theta(m!)$.

6.2. Determine Execution Order Algorithm

The algorithm *DetermineExecutionOrder* is described in Table 8. *DetermineExecutionOrder* determines the target state, between which and s_{start} there is the minimal number of (s_r, c_i)-pairs (i.e., the shortest path $Path^*$, with $cost^* = d^*$). To keep track of the order, in which the states must be produced, the hash table $Predecessor$ is used. $Predecessor$ stores for each s_r, including the target state, the state from which s_r must be produced along the shortest path $Path^*$. $Frontier$ is a min-priority queue. Thus, lines 1 and 13 represent the insert operation. In line 3, the state s_r with the minimum accumulated path cost from s_{start} to s_r is extracted from $Frontier$. In line 12, s'_r is removed from $Frontier$. If $Explored$ is implemented as a linked list, line 4 inserts s_r at the end of the list.

Table 8.The UCS for determining the MEO.

DetermineExecutionOrder(c^*)	
Parameters:	
	$c^* \in C$
Initialization:	
	$Frontier: \mathbb{N}^+_{\leq 2^{m-1}} \to \mathcal{S}_S, Frontier = ()$
	$Explored: \mathbb{N}^+_{\leq 2^{m-1}} \to \mathcal{S}_S, Explored = ()$
	$Predecessor_{s_r} = s_{start}, \forall s_r \in \mathcal{S}_S$
Algorithm:	
1	$Frontier = Frontier \underset{prio}{\uplus} s_{start}$
2	**while** $\lvert Frontier \rvert > 0$
3	$s_r = \min_{prio} Frontier$
4	$Explored = Explored \underset{asc}{\uplus} s_r$
5	**if** $isTarget(s_r, c^*)$
6	**return** *FormatExecutionOrder*($Explored$, $Predecessor$)
7	**for each** $c_i \in \mathcal{A}(s_r)$
8	**if** $\exists s'_r \in Explored(successor(s_r, c_i)$ **is equal to** $s'_r)$
9	**continue**
10	$Predecessor_{successor(s_r, c_i)} = s_r$
11	**if** $\exists s'_r \in Frontier(successor(s_r, c_i)$ **is equal to** $s'_r)$
12	$Frontier = Frontier \backslash s'_r$
13	$Frontier = Frontier \underset{prio}{\uplus} successor(s_r, c_i)$
14	**return** *FormatExecutionOrder*($Explored$, $Predecessor$)

The asymptotic upper bound of the worst-case running time of UCS is $O\left(b^{1+\left\lfloor \frac{cost^*}{\varepsilon} \right\rfloor}\right)$, with $b = \lvert \mathcal{A}(s_r) \rvert$, $cost^* = \sum_{(s_r, c_i) \in Path^*(\mathbb{N}^+_{\leq d^*})} cost(s_r, c_i)$, and $\varepsilon = \min_{(s_r, c_i) \in Path^*(\mathbb{N}^+_{\leq d^*})} cost(s_r, c_i)$. We define $b = m$, because $\lvert \mathcal{A}(s_r) \rvert \leq m, \forall s_r \in \mathcal{S}_S$. Also, we define $\varepsilon = 1$, because $cost(s_r, c_i) = 1,\ \forall s_r \in \mathcal{S}_S, c_i \in C$. Thus, the asymptotic upper bound of the worst-case running time of *DetermineExecutionOrder* is $O(m^{1+d^*})$.

6.3. Format Execution Order Algorithm

The algorithm *FormatExecutionOrder* is described in Table 9. The analysis of the time complexity of *FormatExecutionOrder* is straightforward. The determining factor is the for-loop in line 4. It iterates over the total number of states that are explored by *DetermineExecutionOrder*. In the worst-case, the upper bound of this total number of states is equal to the space complexity of *DetermineExecutionOrder*: $O(m^{1+d^*})$.

Table 9. The formatting of the UCS results.

FormatExecutionOrder($Explored$, $Predecessor$)	
Parameters:	
	$Explored: \mathbb{N}^+_{\leq 2^{m-1}} \to \mathcal{S}_S$
Initialization:	
	$Path^*: \mathbb{N}^+_{\leq d^*} \to \mathcal{S}_S \times C, Path^* = ()$
Algorithm:	
1	$s_r = s_{\lvert Explored \rvert} \in Explored(\mathbb{N}^+_{\leq 2^{m-1}})$
2	$Path^* = Path^* \underset{desc}{\uplus} \left(Predecessor_{s_r}, xand\left(s_r, Predecessor_{s_r}\right)\right)$
3	$predecessor = Predecessor_{s_r}$
4	**for** $l = \lvert Explored \rvert - 1$ **to** 1
5	$s_r = s_l \in Explored(\mathbb{N}^+_{\leq 2^{m-1}})$
6	**if** s_r **is equal to** $predecessor$ **&** s_r **is not** s_{start}
7	$Path^* = Path^* \underset{desc}{\uplus} \left(Predecessor_{s_r}, xand\left(s_r, Predecessor_{s_r}\right)\right)$
8	$predecessor = Predecessor_{s_r}$
9	$cost^* = d^* = \lvert Path^* \rvert$
10	**return** $cost^*, Path^*$

7. Demonstration

To demonstrate the fulfillment of the defined requirements for two exemplary cases, we implemented YUMA in the programming language Go.[3]

In the demonstration, YUMA is represented by a binary file that can be executed by cloud architects on their local machines (e.g., by using a shell). The cloud environment, which the implementation interacts with, is AWS. It uses the AWS software development kit (SDK) to dynamically create and delete EC2 instances. The creation and deletion of the EC2 instances is done as part of the *BuildSearchTree* algorithm. On these EC2 instances, the Ansible roles representing the sub-processes are executed. The Ansible roles must be stored in a specific directory on the same local machine. Hence, cloud architects must clone the (Git) repository containing the required Ansible roles to this directory first. The composition request $c^* \triangleq p^*$ must be specified when executing the binary file as a parameter for the function implementing the *DetermineExecutionOrder* algorithm. The implementation specifies each MEO in the yet another markup language (YAML) and stores them in corresponding Ansible playbooks.

For the example case 2, we implemented a set of Ansible roles that can be composed to a platform service, i.e., a single node Kubernetes cluster. These Ansible roles are described in Table 10. As described in section 2.1, the example case 2 represents an IT service commonly composed by cloud architects in practice.

[3] The logical architecture can be found on GitHub: https://github.com/Floble/go-utils/tree/ucs/algorithms/artificialintelligence/search.

Table 10. The Ansible roles in the demonstration.

Ansible role p_i (c_i)	Description	Dependency
Example case 2: Setup of a single node Kubernetes cluster		
configVM (00001)	Disables local firewall and adds iptables rule.	-
deployPod (00010)	Deploys Flannel to the cluster.	runKubernetes, installDocker
installDocker (00100)	Installs the Docker package.	-
installKubernetes (01000)	Installs the Kubeadm, Kubelet, and Kubectl packages.	-
runKubernetes (10000)	Initializes Kubeadm and creates the cluster config.	installKubernetes

Two kinds of Kubernetes clusters can be composed from the Ansible roles. A blank cluster and a cluster with Flannel as the container network fabric. To setup former, the *runKubernetes* role depending on the *installKubernetes* role must be executed. To setup a Kubernetes cluster with Flannel, the *deployPod* role must be executed. This role deploys a daemonset to an initialized Kubernetes cluster and therefore depends on the *runKubernetes* and *installDocker* roles. When executing the binary file, the YUMA implementation learns the dependencies described in Table 10 autonomously as a result of performing the *BuildSearchTree* algorithm. To determine the MEOs for *runKubernetes* and *deployPod*, the binary representations (see Table 10) of these roles must be passed as parameters to the *DetermineExecutionOrder* algorithm implementation. The MEO for $p^* =$ *runKubernetes* is as following: *installKubernetes* → *runKubernetes*. The MEO for $p^* =$ *deployPod* is as following: *installDocker* → *installKubernetes* → *runKubernetes* → *deployPod*. These MEOs are stored as Ansible playbooks on the local machine.

To simulate the dynamic nature of the cloud environment, for each simulation of the exemplary cases, we added or removed Ansible roles from the directory randomly and executed the *BuildSearchTree* and *DetermineExecutionOrder* algorithms again. Although we executed the algorithms manually, another option would be the implementation of a cronjob executing the binary file in specific time intervals.

8. Evaluation

YUMA has to be evaluated against the requirements defined in Table 6. Proving the fulfillment of R1 is straightforward. As described in section 6, YUMA holds a search tree reflecting the state space and transition model. Regarding the fulfillment of R2 and R3, a more sophisticated proof is required. We demonstrate in section 7 that YUMA fulfills R2 and R3 for two simulated example cases. However, to proof that YUMA fulfills these two requirements for the general case, we must proof the correctness of the algorithms in the learning and inference dimensions of YUMA.

DetermineExecutionOrder is an implementation of UCS. The correctness of UCS has been proven by prior research (Liang & Sadigh, 2019).

8.1. Build Search Tree Evaluation

Preconditions. All sub-processes $p_i \in P$ can be executed (i.e., $\nexists p_i \in P(p_j \rightarrow p_i \wedge p_j \notin P)$).

Postconditions. The algorithm terminates and returns the search tree $\mathcal{M}$ describing all possible paths from s_{start} to s_{end}.

Proof. By induction on $|\mathcal{A}(s_{start})| = |P|$, we prove that the preconditions and execution of the algorithm implies the postconditions.

Base Case. Let $m = |\mathcal{A}(s_{start})| = 0$. Then, $|\mathcal{S}_S| = 1$ and $s_{start} \in \mathcal{S}_S$. In this case, $s_{end} = s_{start}$, because $s_{start} = 2^0 - 1 = 0$. Therefore, the algorithm terminates in line 2 and returns the initialized $\mathcal{M}$ describing only the path from s_{start} to itself (i.e., self-loop). This satisfies the postconditions, because only one path can be described for $m = 0$ (i.e., $0! = 1$).

Inductive Hypothesis. Let $m = |\mathcal{A}(s_{start})|$ and assume that the postconditions hold after executing the algorithm for all $p_i \in P$, which satisfy the preconditions. In the worst-case, the algorithm returns $\mathcal{M}$ describing $m!$ paths from s_{start} to s_{end}.

Inductive Step. Let $m + 1 = |\mathcal{A}(s_{start})|$. Then, $|\mathcal{S}_S| = 2 * 2^m$ and there are $(m+1)m!$ paths to be described by $\mathcal{M}$ in the worst-case. The for-each-loop in line 3 causes the exploration of $m + 1$ states at $d = 1$. For each s_r of these states, the following holds: $|\mathcal{A}(s_r)| = m$. Each such s_r can be considered as s_{start} of a sub-problem with $m = |\mathcal{A}(s_{start})|$ at $d = 1$. Hence, by the inductive hypothesis, from each such s_r there are $m!$ paths to s_{end} in the worst-case. Because there are $m + 1$ such s_r at $d = 1$, in the worst-case, the total number of possible paths from s_{start} to s_{end} is $(m+1)m!$. Therefore, the postconditions are satisfied and, by induction, the algorithm is correct.

8.2. Format Execution Order Evaluation

Proof. A loop invariant is proven to be satisfied at the beginning of every iteration of the for-loop in line 4.

Loop Invariant. At the start of each iteration l of the for-loop, $Path^*$ describes the shortest path from $predecessor$ to the target state determined by *DetermineExecutionOrder*.

Initialization. The loop invariant holds prior to the first iteration of the for-loop. Here, $predecessor$ is the preceding state of the target state as determined by *DetermineExecutionOrder*. Thus, there are no (s_r, c_i)-pairs between $predecessor$ and the target state. In addition, $xand(s_r, Predecessor_{s_r})$ determines $c_i \triangleq p_i$, which must be executed at $predecessor$ to result at the target state. Hence, $Path^*$ describes the shortest path from $predecessor$ to the target state.

Maintenance. To see that each iteration maintains the loop invariant, suppose that $Path^*$ describes the shortest path from $predecessor$ to the target state before the lth iteration. Then, the if-statement in line 6 ensures $s_r = predecessor$. Afterwards, $predecessor$ is defined to be the preceding state of s_r, as determined by *DetermineExecutionOrder*. Because there are no (s_r, c_i)-pairs between $predecessor$ and s_r, $\left(Predecessor_{s_r}, xand(s_r, Predecessor_{s_r})\right)$ describes the shortest path from $predecessor$ to s_r. By our assumption, the adding of this path to $Path^*$ in line 7 results in $Path^*$ describing the shortest path from $predecessor$ to the target state. Thus, incrementing l reestablishes the loop invariant for the next iteration.

Termination. At termination, $l = 0$ and $predecessor = s_{start}$. By the loop invariant, $Path^*$ describes the shortest path from s_{start} to the target state. This is the result that we wanted (i.e., a representation of the MEO of S^*).

9. Discussion and Conclusion

This study contributes to the field of IT service composition in two major ways. First, we are first in formulating the IT service composition problem for the IaC context. While prior studies have formulated the problem for the domains of automated web and cloud service composition (Zou et al., 2010, 2014), our problem definition is inspired by the circumstances that cloud architects must deal with when composing IT services from IaC specifications. We concretize the problem formulation in the form of a search problem. This is appropriate for the IaC context, because our problem formulation focuses on sub-processes and defines them as the primitive actions. Hence, the action space is defined by the set of sub-processes stored in a repository. Subsequently, a state can be represented as the subset of successfully executed sub-processes and the state space and transition model can be reflected by a search tree. This makes it unnecessary to represent the sub-processes and states in PDDL, perform a HTN planning, and build a planning graph.

Second, the related AI planning agents determine the service dependencies based on agent-specific and PDDL domains and problems, and semantic link networks. Therefore, the service dependencies must be specified by the service providers as part of the OWL-S and WSDL specifications provided as input to the agents. As the service dependencies must be known by the service providers, the agents interact with known environments. In contrast, YUMA determines the sub-process dependencies by interacting with a cloud environment in an explorative way. Based on this exploration, it builds its model. Thus, YUMA implements a model learning algorithm enabling it to interact with unknown environments. This makes YUMA more autonomous compared to related agents.

YUMA itself is also an implication for practice. It automates the composition of IT services from IaC specifications without requiring cloud architects to have knowledge about the dependencies between sub-processes stored in a repository. Therefore, it frees up cloud architects from the burden to explore and analyze the IaC specifications of relevant sub-processes and thereby saves them a lot of time. YUMA is expected to reduce the time cloud architects must spend on the composition of an IT service from minutes or even hours to several seconds. Based on our argumentation in section 1, we expect YUMA to create most utility for enterprises in which many sub-processes (e.g., hundreds) are maintained across multiple repositories by not only cloud architects that are experts in IaC. Examples for such enterprises are IT consulting firms.

As with any research, our work comes with limitations that must be addressed by future research. First, the *BuildSearchTree* algorithm that is performed by YUMA to determine the sub-process dependencies must be seen as a bottleneck to the performance. The search tree must be rebuilt regularly with a time complexity of $\Theta(m!)$. Although *BuildSearchTree* is a pure exploratory algorithm and guarantees the discovery of the full state space and transition model, future research can adapt or replace *BuildSearchTree* to incorporate machine learning. Machine learning algorithms such as structured perceptron, *LIVE*, *EXPO*, and *OBSERVER* (Jiménez et al., 2012; Liang & Sadigh, 2019). In addition, for subsequent research, we have started with reformulating the IT service composition problem as a reinforcement learning problem and have experimented with n-step temporal difference learning algorithms to solve it.

Second, YUMA has been demonstrated and evaluated from a formal and conceptual perspective, but from the user perceptions in terms of utility. Future research should evaluate the perspective of cloud architects on the reasoning of the intention to use YUMA. For this, the established models of IS success (e.g., UTAUT2) should be used to conduct a quantitative survey study among cloud architects.

To conclude, YUMA is a first step towards AI-augmented cloud architecture delivery. By automating the IT service composition in the IaC context, YUMA digitizes an important task of cloud architects. As the IaC paradigm has become widely adopted in practice, we encourage other scholars to contribute to this field.

References

Baer, F., & Leyer, M. (2016). Towards assessing the value of digital self-service options from a provider perspective. *AMCIS 2016 Proceedings*.

Becker, J., Beverungen, D. F., & Knackstedt, R. (2009). The challenge of conceptual modeling for product–service systems: Status-quo and perspectives for reference models and modeling languages. *Information Systems and E-Business Management*, *8*(1), 33–66.

Chiari, M., De Pascalis, M., & Pradella, M. (2022). Static Analysis of Infrastructure as Code: A Survey. *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*, 218–225.

Eshuis, R., & Mehandjiev, N. (n.d.). Flexible Construction of Executable Service Compositions from Reusable Semantic Knowledge. *ACM Transactions on the Web*, *10*(1), 27.

Gregor, S., & Hevner, A. (2013). Positioning and Presenting Design Science Research for Maximum Impact. *Management Information Systems Quarterly*, *37*(2), 337–355.

Hatzi, O., Nikolaidou, M., Vrakas, D., Bassiliades, N., Anagnostopoulos, D., & Vlahavas, I. (2015). Semantically Aware Web Service Composition Through AI Planning. *International Journal on Artificial Intelligence Tools*, *24*(01), 1450015.

Jiménez, S., De La Rosa, T., Fernández, S., Fernández, F., & Borrajo, D. (2012). A review of machine learning for automated planning. *The Knowledge Engineering Review*, *27*(4), 433–467.

Johannesson, P., & Perjons, E. (2014). *An Introduction to Design Science* (2014th ed.). Springer.

Jouck, T., & Depaire, B. (2018). Generating Artificial Data for Empirical Analysis of Control-flow Discovery Algorithms: A Process Tree and Log Generator. *Business & Information Systems Engineering*.

Jula, A., Sundararajan, E., & Othman, Z. (2014). Cloud computing service composition: A systematic literature review. *Expert Systems with Applications*, *41*(8), 3809–3824.

Kumara, I., Garriga, M., Romeu, A. U., Di Nucci, D., Palomba, F., Tamburri, D. A., & van den Heuvel, W.-J. (2021). The do's and don'ts of infrastructure code: A systematic gray literature review. *Information and Software Technology*, *137*, 106593.

Kuzu, M., & Cicekli, N. K. (2012). Dynamic planning approach to automated web service composition. *Applied Intelligence*, *36*(1), 1–28.

Liang, P., & Sadigh, D. (2019, Autumn - 2020). *Artificial Intelligence: Principles and Techniques* [Lecture]. Course CS221, Stanford University (Online). https://stanford-cs221.github.io/autumn2019/

Masdari, M., Nozad Bonab, M., & Ozdemir, S. (2021). QoS-driven metaheuristic service composition schemes: A comprehensive overview. *Artificial Intelligence Review*, *54*(5), 3749–3816.

Rao, J., & Su, X. (2005). A Survey of Automated Web Service Composition Methods. In J. Cardoso & A. Sheth (Eds.), *Semantic Web Services and Web Process Composition* (Vol. 3387, pp. 43–54). Springer Berlin Heidelberg.

Razian, M., Fathian, M., Bahsoon, R., Toosi, A. N., & Buyya, R. (2022). Service composition in dynamic environments: A systematic review and future directions. *Journal of Systems and Software*, *188*, 111290.

Russell, S., & Norvig, P. (2016). *Artificial Intelligence: A Modern Approach*. Pearson.

Sampson, S. E. (2012). Visualizing Service Operations. *Journal of Service Research*, *15*(2), 182–198.

Sampson, S. E., & Froehle, C. M. (2006). Foundations and Implications of a Proposed Unified Services Theory. *Production and Operations Management*, *15*(2), 329–343.

Sandobalín, J., Insfran, E., & Abrahão, S. (2020). On the Effectiveness of Tools to Support Infrastructure as Code: Model-Driven Versus Code-Centric. *IEEE Access*, *8*, 17734–17761.

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning, second edition: An Introduction* (2nd edition). Bradford Books.

van der Aalst, W. M. P., Hofstede, A. H. M. ter, Kiepuszewski, B., & Barros, A. P. (2003). Workflow Patterns. *Distributed and Parallel Databases*, *14*(1), 5–51.

Venable, J., Pries-Heje, J., & Baskerville, R. (2012). A Comprehensive Framework for Evaluation in Design Science Research. *Design Science Research in Information Systems. Advances in Theory and Practice*, 423–438.

Yalley, A. A., & Sekhon, H. S. (2014). Service production process: Implications for service productivity. *International Journal of Productivity and Performance Management*, *63*(8), 1012–1030.

Zou, G., Chen, Y., Xiang, Y., Huang, R., & Xu, Y. (2010). AI Planning and Combinatorial Optimization for Web Service Composition in Cloud Computing. *Proceedings of the International Conference on Cloud Computing & Virtualization 2010 CCV 2010*, 28–35.

Zou, G., Gan, Y., Chen, Y., & Zhang, B. (2014). Dynamic composition of Web services using efficient planners in large-scale service repository. *Knowledge-Based Systems*, *62*, 98–112.