A Survey of Web Service Composition Methods and Techniques

ABSTRACT

This paper surveys a branch of the web services domain known as web service composition. We explore three unique subsections: the composability of web services, intelligent web services curation and time-cost relationship involved in web service composition. Following this, we attempt to replicate the results obtained by the authors who analyze the compatibility of web services in web service composition. Using a sample set of their original dataset, we explore how feasible the authors' experiment process was and provide ourselves with a deeper understanding of web service composition.

1 INTRODUCTION

Web Service Composition (Mashup/ Curation) is the process of constructing new web services by combining existing ones [1]. This is often a very challenging process in distributed environments. Web Service Composition emphasizes essential attributes such as location transparency, availability, discovery, and component reusability. The rising adoption of web services has led to an increased number of published services. Single service offerings have become primitive and it is more advantageous to combine multiple services to provide a complex composite service.

The typical stages of service composition are as follows: The initial stage involves composition planning, where the service request is specified and decomposed into an abstract set of tasks. Subsequently, service discovery takes place, involving a search for services that align with both functional and non-functional requirements for each task in the composition. Following the discovery, service selection occurs, aiming to choose the most suitable service for each task to meet user requirements. The final phase, service execution, involves invoking and executing individual tasks in the composition to generate the ultimate service.

The papers that we have chosen to explore delve into most stages of the web service composition process. Papers selected look at the drawbacks and improvements to static web service composition, the cost-time relationship found in the composition process and the compatible API pairs that can potentially inspire the composition of new web services.

2 LITERATURE REVIEW

2.1 Background

We will focus on three unique papers to form the basis of our survey. In this section we briefly discuss these papers, emphasizing their significance to web service composition.

2.1.1 ***Web Service:*** A web service is an interface through which web content such as HTML, JSON, and XML documents can be served via a server.

2.1.2 ***Web Service Composition:*** Web service composition refers to the notion of combining functions of multiple single-service endpoints to deliver a new, more powerful service. This advantageous advancement in the web technologies field has enabled faster development of deployment of integrated API services. The concept of web service composition has been around for a little more than a decade, however a significant number of research topics have been explored thanks to the increased pervasiveness and accessibility of the internet.

2.2 Paper 1 [6]

2.2.1 ***Targeted Research Issues:*** As web applications become increasingly prevalent in today's internet world, more developers are integrating Web APIs into their programs than ever before. By leveraging prefabricated APIs, developers can not only prototype software applications in a shorter amount of time but potentially integrate multiple API functions into a new feature. Nonetheless, communication among distinct APIs can be tricky - especially when their schemas and syntax do not match with one another. For this reason, this paper attempts to investigate in detail whether multiple public Web APIs can be combined to form new services and the levels of compatibility those APIs exhibit. According to the paper, two API schemas are considered compatible if they require little or no manipulations of their associated properties to compose a new service. An example of performing a minor manipulation is the conversion from a number to a string, which does not involve any additional computation using the data's original value.

2.2.2 ***Technical Contribution of Paper:*** This paper contributes to the research of web service composition in a number of aspects. First, it is the first attempt to systematically and statically evaluate the compatibility of API schemas.

From a high-level perspective, the paper's experimental results have successfully answered some of the most critical questions related OpenAPI-compatible endpoints. First, the paper provided aggregated compatibility statistics at the property and the data type levels. These number suggest that the more compatibility levels are considered the less likely there will be an endpoint or API match. Second, after classifying compatibility levels for each API and endpoint, it was determined that the compatibility at the property name level is lower than that of the data type level, and when both levels are factored into the analysis process they can greatly reduce the overall ratio of compatible APIs. Third, using the probability developed in the paper, it was concluded the more endpoints an API defines, the more likely it has a compatible counterpart. Last but not least, endpoints on average have compatibility rates of 37% and 72% for sink and source, respectively. This once again resonates with the goal to precisely compute the extent to which API endpoints are compatible.

This paper lays the foundation for future applications such as automatic API compatibility validation tools, API recommendation systems for compatible API endpoints, and middleware that facilitate communication among APIs with incompatible schemas.

2.2.3 ***Methodology and Results:*** The paper **"How Composable is the Web? An Empirical Study on OpenAPI Data model Compatibility"** systematically examines approximately 20,000 OpenAPI Specification-based endpoints to determine the likelihood of two or more APIs being integrated to form a new web service.

Specifically, the study attempts to address questions including the compatibility of APIs on the schema level, whether or not two more APIs can directly support one another's format, and the number of compatible API pairs that can be formed from the pool of selected endpoints. The analysis is carried out in several steps. First, the researchers scrape APIs that are available to the public on the internet, as well as a dataset from a previous research project. They then develop a set of rules to validate the schemas of APIs' formats, and filter out those that do not pass the validity check. Subsequently, schemas that have been determined valid are extracted for further processing. Because of their low usage, schemas containing any "not" operators are removed. Finally, after all the pre-processing of the raw data, they perform a series of compatibility analyses and algorithms to narrow down the set of potentially compatible API endpoints.

In the initial data collection and preparation phase, several factors are considered to filter out invalid schemas. First, a valid schema should always contain an "OAS" or "Swagger" field to correctly identify the OpenAPI version. Second, the schema ought to include a "title" in its "info" section. Third, those schemas with a proper server and base path field are discarded. Lastly, any acceptable schema must have at least one endpoint that ingests or outputs data with a minimum of three endpoint paths.

The set of rules used to validate the schemas have been developed in accordance with the supported data types for each schema property. There are 16 properties that have been directly adopted from the JSON Schema Specification. These include: title, multipleOf, maximum, exclusiveMaximum, minimum, exclusiveMinimum, maxLength, minLength, pattern, maxItems, minItems, uniqueItems, maxProperties, minProperties, required, enum. This set is denoted as P-J. Additional property keywords from the JSON Schema Specification have also been modified and adopted to support more complex capabilities within OpenAPI definitions. They are: type, allOf, oneOf, anyOf, not, items, properties, "additionalProperties", description, format, default. This other set is defined as P-A. At the top-most level of the schema, each key must conform to the following rules to be valid. First, if the key is in set P-J, excluding title, pattern, required, and enum, its associated value must be a number. Second, if the key is title, pattern, type, format, or description, the value must be a string. Third, if the key is either required or enum, its value should be a simple array. Four, if the key belongs to set P-A except type, format, and description, the value must be a complex array. Finally, if the key is properties or items field, its value should only contain an object.

2.2.4 ***How This Paper Compares to Existing Solutions:*** Several past research projects have provided guidance for this paper. For example, this paper takes a similar approach as the work "Composing JSON-Based Web APIs " [3] in examining schema element names and types. It is also related to the paper "An Empirical Study of GraphQL Schemas ". [8] While the paper has discovered a number of naming convention issues and denial-of-service security risks, it does not go on the investigate the compatibility of GraphGL API schemas. Another paper that is relevant to this area of research is "Web APIs Structures and Data Models Analysis". Still, it only measures the size of API structures and their data model schemas instead of the compatibility challenges discussed in this paper. [5]

2.3 Paper 2 [10]

"Towards an Adaptive Curation Services Composition based on machine learning"[10] describes an approach to adaptive web service composition.

2.3.1 ***Targeted Research Issues:*** Curation services are algorithms or processes used to clean, enrich, transform, or otherwise prepare data for analysis or consumption. They aim to improve the quality, usability, and relevance of data. Curation services encompass a wide range of functionalities, such as data cleaning, normalization, entity extraction, sentiment analysis, etc. Each curation service performs a specific data processing task to enhance the data's value or usability. As such, they can be implemented using various technologies and algorithms, depending on the specific task they perform.

There is generally a great dependency on the generated outcome from web service curation composition and the inputs. These inputs can either be functional such as the source of input data or non-functional such as the decision context and user constraints. While traditional static web service composition can meet the user need of functionality, when the environment being the component service, or user requirement changes, the composition must be manually updated to adapt to a new need. For example, consider a scenario where a curation composition service is used to recommend personalized content to users based on their preferences and browsing history. The service employs workflow-based composition to select and present content to users. However, if there's a sudden change in user behavior or preferences, such as a trending topic or event, the fixed workflow may struggle to adapt. This could be due to hard-coded and predefined paths. For example, during a major news event, users may suddenly show interest in a specific topic, causing a surge in demand for related content. The authors propose an original approach "Adaptive CUration Service Composition (ACUSEC)" that can perform adaptive context aware and user-oriented data curation.

2.3.2 ***Technical Contribution of Paper:*** ACUSEC aims to dynamically compose curation services based on various factors such as user preferences, content characteristics, and context.

The technical contribution of the paper lies in the development of ACUSEC, an approach for adaptive and context-aware data curation. ACUSEC integrates a library of curation services, including **extraction services** for NLP tasks like named entity extraction, **enrichment services** for semantic enrichment, **data quality control services** for detecting missing values and anomalies, and **data standardization services** for unifying data using a knowledge base. The key innovation of ACUSEC is its adaptive curation service composition, which occurs in two stages: training and composition.

In the training stage, a Markov Decision Process (WSC-MDP) is employed to represent all valid curation service compositions. A Markov Decision Process (MDP) is a mathematical framework used to model decision-making processes in situations where outcomes are partially random and partially under the control of a decision maker. "WSC-MDP" then stands for "Web Service Composition Markov Decision Process", a specific type of MDP tailored for modeling the process of composing web services to achieve an objective. In WSC-MDP, states represent different configurations

or compositions of web services, actions correspond to selecting or executing particular web services, transition probabilities are the likelihoods of moving from one state (composition) to another based on the selected actions, and rewards represent the benefits or costs associated with each state-action pair.

The Q-Learning algorithm is then used to learn weights (rewards) assigned to each service based on QoS, data source type, decision context, and user preferences and constraints. QoS stands for "Quality of Service". The QoS refers to the set of characteristics that describe the performance and reliability of a service. This could include the price, success rate, availability etc. The learned weights are stored in a Q-Table, which represents the optimal policy for service composition. In the composition stage, "Algorithm 1" is executed, using the learned Q-Table to select the most suitable curation services composition based on the desired features.

"Algorithm 1" is a composition algorithm that iteratively selects the next state (i.e., curation service) from the Q-Table, prioritizing states with the maximum reward. This algorithm continues until either all available states are visited or the desired service is created, resulting in a curated data set tailored to the specific context and user requirements.

2.3.3 ***Methodology & Results:*** The authors' evaluation of ACUSEC begins with an examination of reinforcement learning performance, focusing on the execution time of the entire learning process for service composition using a library of curation services. The authors use two data sources, which vary in their data source structures and complexities, to assess ACUSEC's effectiveness. One source represents COVID-19 genomes from NCBI (National Center for Biotechnology Information) and the other is of clinical data on the flu from FluPrint. They test ACUSEC approach's performance in composing curation services for different data source types as the COVID-19 genome dataset is semi-structured data and the FluPrint dataset is of structured data.

Results show that while execution time remains low for smaller Q-Table sizes, it increases as the size grows. The training algorithm also demonstrated quick convergence after 400 episodes. Additionally, for both the training and composition algorithms are found to have a linear time complexity, which is favourable for scaling.

The authors also test ACUSEC's adaptability to changes in data source type, user preferences and constraints, and Quality of Service (QoS) values.

(1) **Adaptability to Data Source Type**
- **Process:**
  (a) Defined different data source types: structured and unstructured/semi-structured.
  (b) Set user preferences equal for all QoS values.
  (c) Performed curation service composition for each data source type.
- **Results:**
  (a) ACUSEC successfully distinguished which curation services were suitable for each data source type based on their structure.
  (b) Services applicable only to specific data structures (e.g., structured vs. unstructured) were included/excluded accordingly.

(2) **Adaptability to User Preferences**
- **Process:**
  (a) Defined user preferences with specific weights for each QoS dimension:
    – Accuracy: 10%
    – Availability: 10%
    – Reliability: 50% (highest weight)
    – Response Time: 10%
    – Reputation: 10%
    – Security: 10%
  (b) Performed curation service composition for a structured data source.
- **Results:**
  (a) Composition differed from the one obtained with equal preferences for all QoS dimensions.
  (b) User preferences influenced the chosen services.
  (c) Example: Entity extraction service was replaced by stem extraction service due to its higher reliability (90% vs. 80%).

(3) **Adaptability to User Preferences - Constraint Influence**
- **Process:**
  (a) Defined a user constraint: Accuracy $\geq$ 80% for a structured data source.
  (b) Maintained equal preferences for all QoS dimensions.
  (c) Performed curation service composition with the constraint.
- **Results:**
  (a) Composition differed from the one without constraints.
  (b) Services not meeting the accuracy constraint (e.g., rules extraction and linking services with accuracy 76% and 71% respectively) were excluded.

(4) **Adaptability to QoS Changes**
- **Process:**
  (a) Assigned random QoS values to curation services.
  (b) Performed multiple curation service compositions for each data source structure with these varying QoS values.
- **Results:**
  (a) ACUSEC considered the changing QoS values.
  (b) The generated service composition adapted to the newly assigned QoS values, demonstrating flexibility (the authors do not expand on this).

(5) **Adaptability in Critical Decision Context**
- **Process:**
  (a) Simulated curation for semi-structured data sources using a static pipeline (Stem Extraction, Synonym Extraction, Linking Services).
  (b) Assumed user constraints: Accuracy > 75% and Response Time > 70%.
  (c) Used ACUSEC with these defined constraints.
- **Results:**

(a) ACUSEC generated a composition that met the constraints: Stem Extraction → Synonym Extraction Service.
(b) Linking service, although having a good response time, was excluded due to not meeting the accuracy constraint.

2.3.4 ***How This Paper Compares to Existing Solutions:*** The paper presents a novel approach to adaptive service composition, emphasizing the consideration of various factors such as QoS, user preferences, decision context, and user constraints simultaneously. While existing solutions like those proposed by Yan et al.[9] and Yang et al.[4] focus on dynamic adaptation based on either QoS values or user preferences, the approach introduced in this paper integrates multiple criteria for composition decision-making. Moreover, while Wang et al.'s[7] work also employs reinforcement learning for adaptation, it primarily addresses composite service adaptation rather than comprehensive data curation needs. In contrast, the approach in this paper specifically targets intelligent data curation, addressing challenges posed by multi-structured data sources. These factors distinguishes the paper's contribution as it offers a more holistic and flexible framework for adaptive service composition, particularly suited for complex data curation tasks in diverse environments.

2.3.5 ***Example - Adaptive Healthcare Data Curation:*** Consider a healthcare organization tasked with aggregating and analyzing diverse datasets from various sources, including clinical records, genetic data, and patient-generated health data, like wearable devices. Traditional static methods of data curation struggle to adapt to the evolving needs and complexities of such datasets. However, an adaptive approach like ACUSEC could revolutionize healthcare data curation:

(1) **Dynamic Data Sources:** ACUSEC can dynamically compose curation services based on the changing nature of healthcare data sources. For instance, if new genetic sequencing technologies emerge, ACUSEC can adapt its curation strategies to incorporate these novel data types seamlessly.
(2) **User Preferences and Constraints:** Healthcare professionals may have specific preferences regarding data formats, privacy concerns, and analytical requirements. ACUSEC can tailor the curation process to accommodate these preferences while ensuring compliance with regulatory constraints such as HIPAA.
(3) **Quality of Service (QoS) Considerations:** Healthcare data often comes with stringent quality requirements, including accuracy, completeness, and timeliness. ACUSEC can prioritize curation services based on QoS metrics to ensure that high-quality data sets are produced for clinical analysis and decision-making.
(4) **Context-Aware Composition:** In healthcare, the context of data usage can vary significantly, from clinical research to patient care delivery. ACUSEC's adaptive composition framework can dynamically adjust to different contexts, optimizing the curation process for specific users.

By employing ACUSEC in healthcare data curation, organizations can enhance their ability to derive actionable insights, improve patient outcomes, and drive innovation in healthcare delivery.

2.4 Paper 3 [2]

The paper titled **"Towards Representing Time-Cost Tradeoffs for Service Compositions"**[2] by Franziska S. Hollauf, Marco Franceschetti, and Johann Eder addresses the need for a more nuanced representation of the relationship between time and cost in service compositions.

2.4.1 ***Targeted Research Issues:*** The authors argue that traditional Service Level Agreements (SLAs) may not adequately capture the dynamics of this relationship which may not always be monotonous, particularly when considering the actual time taken by services to fulfill requests.

To address this shortcoming, the paper introduces an algorithm designed to compute a time-cost table, which offers a more detailed understanding of how time and cost interact within service compositions. This table serves as a foundation for optimizing composed services according to client demands and budget constraints. Section IV of the paper in particular presents all the operations used to construct the algorithm.

By focusing on the temporal aspects of SLAs, the authors aim to contribute to ongoing research in service composition and enable finer tuning of requirements, which can potentially leverage machine learning techniques discussed in Paper 2 of our literature review to construct composite services that satisfy the users exact needs while abiding to their budgetary constraints.

2.4.2 ***Technical Contribution of Paper:***
Defining and Representing Service Compositions -
To understand service compositions, the authors propose a new way to visualize the structure of composite services in the form of **composition trees**. The authors go on to discuss a few components that go into describing a composition tree. These include leaf nodes that denote individual services and inner nodes that act as *constructors* which provide pathways to more constructors or basic services. Constructors can be thought of as conditional operators that decide which service to invoke (SEQ, XOR, XORC and XORD) or parallelized execution of two services at once (AND). Figure 1 from the authors can be used as an example to understand the concept of composition tree -

Constructors -
Figure 1 represents the composition tree using constructors to denote the logical flow of operations. There are 4 kinds of constructors -

(1) AND - This construct denotes the scenario where two services can run in parallel (Data Integration and Logging service).
(2) SEQ - This construct denotes a sequential execution of operations in the composite service (Data Integration runs after Data Storing Service is finished).
(3) XORC - Conditional alternative is one of the alternative constructors where the choice between alternative sub-services depends on evaluating a condition during operation.

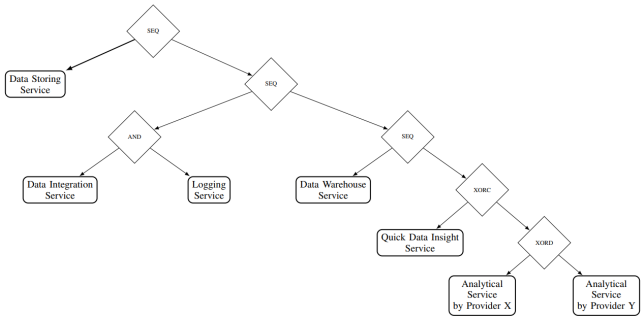

Figure 1: Composition Tree for a service to analyze business data using a BPMN diagram[2]

(4) XORD - Decisional alternative is the other alternative constructor where the choice between alternative sub-services is made by the controller.

Time-Cost Tables -
To tackle the main problem explored through this literature the authors propose a data structure they denote as the Time-Cost table. The time-cost table is used to represent the relationship and in turn the trade-off between the time and cost of a service. For composed services it can be used to properly select services with respect to monetary and temporal constraints. Each node in the above composition tree representing a service will have its own associated time-cost table.

In its basic form, the authors describe the time-cost table as **a tuple *(f, t, c)*** which represents the duration d for an activity between minimum f and maximum t time units (f≤ d ≤ t) which costs c units[2]. The time-cost table is in place to simply denote the temporal-monetary relationship. The authors do not assume it to signify any service to be faster or any service to be cheaper, as there could be services which are faster and better but cost more and vice versa depending purely on a business perspective and use-case.

2.4.3 *Methodology & Results*.

The authors elaborate on the assumptions they chose to make when computing time-cost tables along with every operation involved in calculating the time-cost tables in detail. The following operations are discussed by the authors with a brief summary for each operation -

(1) Addition of Cost Values of Time-Cost table - Denotes the addition of cost function in the time-cost table. If c and c' are cost values -

$$c + c' = \begin{cases} c + c', & \text{if } c, c' \neq null \\ null, & \text{otherwise} \end{cases}$$

(2) Min/Max of a Set of Cost Values - The min of a set of cost values is the minimum of its *non null values*. The max of a set of cost values is the maximum of its values if there is no null value; or the null value otherwise.

A		B		A_{align}		B_{align}	
duration	cost	duration	cost	duration	cost	duration	cost
10 - 30	10	20 - 40	9	10 - 20	10	10 - 20	null
30 - 60	20	40 - 60	15	20 - 30	10	20 - 30	9
				30 - 40	20	30 - 40	9
				40 - 60	20	40 - 60	15

Table 1: Time-cost table for alignment operation

(3) Selection - The selection function σ is defined as selecting the cost for a given service in a time period from the time-cost table. Σ is defined as the selection of a set of cost values for a given duration.

(4) Point Set - This operation signifies choosing the discrete values for durations from the time-cost table. For example, in **??** the point set would be = [5, 15, 30].

(5) Make Interval - This operation creates a set of continuous intervals from a set of durations D. For example if D = [1,5,9, 18], The Intervals I = [(1,5), (5,9), (9, 18)]. This function thus creates a dense set of duration intervals.

(6) Alignment - This operation makes it easier to calculate time-cost tables in presence of the AND and XOR operators discussed previously. It allows to expand the time intervals of two tables such that they overlap as shown in Table 1.

(7) Compression & Normalization - Compression function takes a time-cost table and converts it to a dense, functional time-cost table. Normalization minimizes a functional time-cost table by joining adjacent intervals with same cost and making them continuous. A few additional definitions to supplement the discussion -
   (a) Functional - A time-cost table is said to be functional if the intervals **do not overlap**.
   (b) Dense - The authors describe a dense time-cost table as a table where there exists a cost for every interval within the range of the table. This implies that the intervals are continuous.

Calculating Time-Cost Tables - In this section the authors discuss how to calculate the time-cost table S for a composite service which is has individual services with time-cost tables A & B by using any of the previously discussed service constructors (SEQ, EXT, AND, XORC & XORD). Time-cost table are constructed using Composition Trees. Thus its a bottom-up approach as calculating the table for a composite service requires the tables for its child nodes/sub-services to be calculated already.

For demonstration purposes we use original examples here to best represent the concepts discussed by the authors.

(1) Sequence (SEQ) -
Mathematically - Let A, B be functional time-cost tables, $A + B = SEQ(A,B) = \forall$ *(f, t, c), (f', t', c')* $\in T : f \leq f' \rightarrow c \leq c'$.
In simpler terms, the authors define calculating the time-cost table for a SEQ constructor by adding the time-intervals from tables A and B and then adding the *correlated costs*

A		B		Intermediate		A + B	
duration	cost	duration	cost	duration	cost	duration	cost
1 - 10	5	1 - 30	20	2 - 40	25	2 - 11	25
10 - 20	15	30 - 40	15	31 - 50	15	11 - 21	25
20 - 25	30			11 - 50	35	21 - 31	25
				40 - 60	30	31 - 40	15
				21 - 55	50	40 - 50	15
				50 - 65	45	50 - 55	30
						55 - 60	30
						60 - 65	45

Table 2: Time cost table for SEQ Constructor

so that the new intervals get the minimal costs assigned. Table 2 shows the table calculations, the Intermediate table is calculated by adding intervals and the final table is calculated by applying Compression and making intervals continuous.

(2) Extension (EXT) - Extension function simply adds intervals to extend the time-cost table adding the possibility to *waiting up to d units*. For example - Original duration [(1-3), (3-10)] become [(1,3), (3-7), (7-14)] if we add the "extension" to wait for 4 units between the original two intervals. Please note how the original duration length of 7 units is maintained in the new interval (7-14).

(3) Parallel Execution (AND) - Parallel execution introduces the assumption that the intervals for all the services are the same. In such cases a time-cost table can be constructed by keeping the intervals intact and just adding the costs as shown in Table 3. In case the intervals are staggered,

Service A		Service B		A ∧ B	
duration	cost	duration	cost	duration	cost
10 - 20	5	10 - 20	8	10 - 20	13
20 - 40	15	20 - 40	10	20 - 40	25

Table 3: Time cost table for AND Constructor

the ***AND+*** constructor will add wait activities which help in making intervals continuous using the EXT operator as shown in Table 4.

Service A		Service B		A ∧ B	
duration	cost	duration	cost	duration	cost
10 - 20	4.40	1 -25	8	1 - 10	8
20 - 40	8.80	25 - 50	16	10 - 20	12.40
				20 - 25	16.80
				25 - 70	24.80
				75 - 90	null
				90 - 100	null

Table 4: Time cost table for AND+ Constructor

(4) Alternatives (XOR) - The XOR constructor has two pathways - XORC and XORD. For either constructors the intervals are partitioned first using the alignment operation (refer to Table 1.
XORC chooses one of two services based on the evaluation of a condition. As its impossible to know the output for the condition evaluation beforehand, **worst case cost is assumed when using XORC**. Thus for each duration interval the worst cost from the two individual time-cost tables is considered for the new composite table.
XORD operator leverages the decision making capability of the controller to choose one of the two services. The assumption here is that the controller **always selects the service with the lower cost** for a given duration. By above definitions, from Table 1 XORD will choose Service B while XORC will choose Service D.

*2.4.4 **How This Paper Compares to Existing Solutions**.*

The authors present a novel approach to realise and evaluate the relationship between the time and cost for consumption of web-services. As of this review we are not aware of any other existing and different approaches to evaluate the problem statement the authors present. For the most part a majority of approaches to review this problem assume a monotonic approach as highlighted by the authors. This approach helps answer questions regarding what service to choose given limitations of time and monetary budgets. The granularity of this approach provides insight in choosing services that build large composite services.

3 METHODOLOGY

We followed and implemented the steps as instructed by Paper 1. The following subsections will explore each of these steps in detail.

3.1 Experimental Setup

All of our data processing work has been completed in Python scripts. These scripts can be run on any commodity workstation that provisions Python 3.9 or above and they do not consume many computational resources. Our script requires a stable internet connection to acquire the raw sample files for the initial stage, while later tasks do not require networking but demand 1GB of permanent storage space for downloaded files.

3.2 Sample Collection

The authors acquire their data from multiple sources (SwaggerHub, APIs.guru, BigQuery and from scanning the internet) as stated in the blog post associated with the paper. We have however limited our experiment to the data from APIs.guru and scanning the internet to make the scope of the project more manageable. We therefore had a collection of 71,415 API descriptions.

3.3 Data Preparation and Validation

The data preparation is performed while adhering to the constraints of the authors. After initial collection of the data we validate each API Specification file against the following constraints as the first part of filtering out invalid schemas which would bias our results -

(1) **GET/POST Parameter check** - APIs need to have at least one endpoint with a description of schema of consumed (GET) or produced data (POST). To satisfy this condition we scan through the data and check if there is a 'get' or 'post' request in the paths. This condition ensures that the results are not biased towards mismatches as there can be requests which are just notification requests.
(2) **Valid Endpoints** - APIs need to have at least 3 paths in order not to bias the average composability rate. To satisfy this condition we scan through the data and check if there are at least 3 paths in the API description.
(3) **Realistic URL** - APIs need to have at least one realistic server (OAS v3) and base path (Swagger) URL. To satisfy this condition we scan through the data and check if there is a 'host' or 'servers' key in the data. We have continued the data cleaning with the assumption that if a base-path key exists its default value is held to be '/'.
(4) **Comparable Schema Objects** - Due to distinct technical requirements, public API definitions can present themselves in inconsistent formats. This stage of the data preprocessing pipeline is to ensure each keyword defined in a schema object strictly conforms to their specific data type. We precisely followed the validation rules as described in the original paper and wrote an experimental script that enforces these rules against schema objects. However, as applying these checks to datasets at a large scale requires significant computational resources, we plan to incorporate this script into the core algorithm as part of our future work.

3.4 Canonical Form Transformation and Feature extraction

The goal of canonical form transformation is to repurpose the default JSON schemas into a unified representation which is compatible for comparison with other schemas. In other words, in case of a complexType parameter, it becomes unrealistic to compare schemas in their default representation. **Traversing the schema using the $ref attribute to get the primitive type** of the parameter and repurposing the schema into a unified representation would allow a fairer comparison.

We couple canonical form transformation and feature extraction in our experiment by creating unified representations of GET and POST operations for each schema's original representation. Using python and the json library we resort to a primitive dictionary data structure to represent the simpler form. We aim to eventually store these in a document database (MongoDB). For our initial experiments, the MongoDB cursor did not stay open long enough to traverse our entire dataset. Therefore, the simpler forms were stored in two folders (each for GET and POST decriptions) within the local repository.

The authors also describe a list of schema properties that do not contribute to the matching process, hence when constructing the unified representation we choose to ignore these properties for operational and efficiency purposes. These include - *description, title, default, example, readOnly, writeOnly, example, title, enum, required, minimum, maximum, multipleOf, minItems, maxItems.*

After construction the unified representation will have the following format -

(1) **GET Operation representation for an API schema**

```
api_path {
    get_operationNumber_parameters {
        parameter_1 : value,
        parameter_2 : value,
        parameter_3 : value
    }
}
```

(2) **POST Operation representation for an API schema**

```
api_path {
    post_operationNumber_parameters {
        parameter_1 : value,
        parameter_2 : value,
        parameter_3 : value
    }
}
```

After cleaning the dataset and transforming the JSON schemas, our overall collection was reduced to 2,908 API descriptions.

3.5 Compatibility Analysis

The authors describe several match types that they use to judge how compatible two API descriptions are. We have carries out data-type matching and property-name matching. We have considered that the dataflow between APIs moves from a post request to a get request. Therefore, we compare all post operations against the get operations to determine matches.

3.5.1 Data-Type Matching. For two APIs to have a data-type match, the keys in which they share must have the same type. For example, in the description below, the 1st post operation matches with the 1st and 3rd get operations. It does not match with the 1st get operation because the datatype of age is float. The 2nd post operation has one match, being the 2nd get operation.

```
api_doc_1  = {
    person_details {
        post_operation_get_parameters {
            name : string,
            age : int,
            height : float
        }

        post_operation_post_parameters {
            name : string,
            age : float
        }
}
api_doc_2  = {
    person_details {
        get_operation_get_parameters {
            name : string,
            age : int,
            height : float
        }
```

```
        get_operation_get_parameters {
            name : string,
            age : float,
            height : float
        }

        get_operation_post_parameters {
            name : string,
            age : int
        }
}
```

3.5.2 Property-Name Matching. For two APIs to have a property-name match, the keys must be exactly the same. Here, we do not consider the values, aka, the data types of the keys. For example, in the description below, there is only one property-name match being that the 2nd post operation matches the 3rd get operation.

```
api_doc_1  = {
    person_details {
        post_operation_get_parameters {
            name : string,
            age : int,
            in : query
        }

        post_operation_post_parameters {
            name : string,
            age : float
        }
}
api_doc_2  = {
    person_details {
        get_operation_get_parameters {
            name : string,
            age : int,
            height : float
        }

        get_operation_get_parameters {
            name : string,
            age : float,
            height : float
        }

        get_operation_post_parameters {
            name : string,
            age : int
        }
}
```

4 ANALYSIS

(1) *Are most APIs compatible on a syntactic or PropertyName Level?* After our analysis, we found that 54.04% of APIs are compatible. On the contrary, the authors found 92.6% of APIs to be compatible. This could be attributed to the fact that we did not use the entire dataset as the authors did.

(2) *How compatible are Web APIs?* While we plan to fully answer this question by determining the percentage of matches that are formed on both data-type match and property-name match, we can for now say that at least half of the API's are compatible on a property-name match. This would still require manipulation of the data types in the middleware.

(3) *On which level are most API compatible* The authors claim that there are less APIs which use the same property names than the ones that share the same property type. We have come to the opposite conclusion as 54.04% were compatible on the property-name level and only 0.31% were compatible on the data-type level. Additionally, we make the claim that APIs are most compatible on the first level of their parameter descriptions. The deeper we go into the document, we would risk losing some property level matches.

(4) *What is the likelihood that a random pair of APIs has at least one compatible endpoint?* Though the initial sample size was large, after cleaned we only used 2,908 API descriptions. From this, we then extracted 58,417 GET operations and 240,354 POST operations. This is a relatively small sample size and given the small number of data-type matches, the number of APIs that satisfy both types of matches is likely to be small. Therefore, the likelihood of a random pair of APIs having at least one compatible endpoint is not zero, but it is low. This is only the result from our experiment, and does not necessarily negate the work and findings of the original authors.

5 DISCUSSION

In this section, we provide a brief discussion of the 3 papers explored in this literature review.

5.1 Paper 1: "How Composable is the Web? An Empirical Study on OpenAPI Data model Compatibility"

- **Targeted Research Issues:** The paper addresses the challenge of combining multiple public Web APIs to form new services and assesses the compatibility levels among them.
- **Technical Contribution:** It lays the groundwork for future tools and systems for automatic API compatibility validation, recommendation systems for compatible API endpoints, and middleware for communication among APIs with incompatible schemas.
- **Methodology & Results:** The study systematically examines OpenAPI/Swagger Specification-based endpoints to determine their compatibility. It develops rules to validate schemas, filters out invalid ones, and performs compatibility analyses. Results suggest a substantial number of compatible API pairs, paving the way for potential mashups.
- **Comparison to Existing Solutions:** This paper uniquely focuses on API compatibility and offers insights into schema-level compatibility challenges.

5.2 Paper 2: "Towards an Adaptive Curation Services Composition based on machine learning"

- **Targeted Research Issues:** Addressing the challenge of static web service composition, the paper introduces ACUSEC, an approach for adaptive and context-aware data curation.
- **Technical Contribution:** ACUSEC dynamically composes curation services based on factors like user preferences, content characteristics, and context. It utilizes reinforcement learning and a Markov Decision Process (WSC-MDP) for adaptive composition.
- **Methodology & Results:** Evaluation involves examining reinforcement learning performance and assessing adaptability to changes in data source type, user preferences, and QoS values. Results demonstrate effective adaptation and scalability, particularly in handling dynamic contexts.
- **Comparison to Existing Solutions:** This paper stands out for its approach to adaptive service composition, integrating multiple criteria for decision-making and focusing on intelligent data curation.

5.3 Paper 3: "Towards Representing Time-Cost Tradeoffs for Service Compositions"

- **Targeted Research Issues:** The paper addresses limitations in traditional Service Level Agreements for capturing dynamic relationships between time and cost in service compositions.
- **Technical Contribution:** It introduces a novel algorithm for computing time-cost tables to represent the time-cost relationship within service compositions. The approach offers a nuanced understanding of time and cost dynamics.
- **Methodology & Results:** The paper elaborates on the algorithm for computing time-cost tables and discusses various operations involved. The evaluation involves constructing time-cost tables for composite services and demonstrating the effectiveness of the proposed approach.
- **Comparison to Existing Solutions:** Unlike existing approaches, this paper provides a detailed framework for understanding and optimizing time-cost tradeoffs in service compositions.

5.4 Overall Comparison

- **Diverse Focus:** Each paper addresses distinct aspects of web service composition, ranging from API compatibility to adaptive composition and time-cost tradeoffs.
- **Methodological Rigor:** All papers employ systematic methodologies, including empirical studies, algorithm development, and evaluation to validate their contributions.
- **Innovative Solutions:** Each paper introduces novel approaches or algorithms to tackle specific challenges in web service composition.
- **Practical Implications:** The findings of these papers hold significant implications for industry practitioners, offering insights and tools for improving the efficiency and effectiveness of web service composition processes.

6 CONCLUSION

In this paper, we have reviewed three recent publications in the web composition domain. We then focused on the first paper and implemented a simplified version of the compatibility detection framework as proposed in the paper. Due to the lack of access to the full set of Web API endpoints from the original work, our results indicated a mismatch in the compatibility ratio between what was computed in our experiment and the original project. Yet we stand by the conclusion that this is a robust approach to determine the composability of web services and in tandem with neural networks and time-cost tables can provide a good framework for large scale composability and analysis of composite services.

REFERENCES

[1] Aram AlSedrani and Touir Ameur. 2016. Web Service Composition Processes: A Comparative Study. *International Journal on Web Service Computing* 7 (03 2016), 1–21. https://doi.org/10.5121/ijwsc.2016.7101

[2] Franziska S. Hollauf, Marco Franceschetti, and Johann Eder. 2021. Towards Representing Time-Cost Tradeoffs for Service Compositions. In *2021 IEEE International Conference on Services Computing (SCC)*. 79–88. https://doi.org/10.1109/SCC53864.2021.00020

[3] Javier Luis Cánovas Izquierdo and Jordi Cabot. 2014. Composing JSON-Based Web APIs. In *Web Engineering*, Sven Casteleyn, Gustavo Rossi, and Marco Winckler (Eds.). Springer International Publishing, Cham, 390–399.

[4] Sangwon Lee and Kwang-Kyu Seo. 2016. A Hybrid Multi-Criteria Decision-Making Model for a Cloud Service Selection Problem Using BSC, Fuzzy Delphi Method and Fuzzy AHP. *Wirel. Pers. Commun.* 86, 1 (jan 2016), 57–75. https://doi.org/10.1007/s11277-015-2976-z

[5] Souhaila Serbout, Fabio Di Lauro, and Cesare Pautasso. 2022. Web APIs Structures and Data Models Analysis. In *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*. 84–91. https://doi.org/10.1109/ICSA-C54293.2022.00059

[6] Souhaila Serbout, Cesare Pautasso, and Uwe Zdun. 2022. How Composable is the Web? An Empirical Study on OpenAPI Data model Compatibility. In *2022 IEEE International Conference on Web Services (ICWS)*. 415–424. https://doi.org/10.1109/ICWS55610.2022.00068

[7] Hongbing Wang, Xuan Zhou, Xiang Zhou, Weihong Liu, Wenya Li, and Athman Bouguettaya. 2010. Adaptive Service Composition Based on Reinforcement Learning. *Service-Oriented Computing* 6470, 92–107. https://doi.org/10.1007/978-3-642-17358-5_7

[8] Erik Wittern, Alan Cha, James C. Davis, Guillaume Baudart, and Louis Mandel. 2019. An Empirical Study of GraphQL Schemas. arXiv:1907.13012 [cs.SE]

[9] Yuhong Yan, Pascal Poizat, and Ludeng Zhao. 2010. Self-Adaptive Service Composition Through Graphplan Repair. In *2010 IEEE International Conference on Web Services*. 624–627. https://doi.org/10.1109/ICWS.2010.91

[10] Firas Zouari, Chirine Ghedira-Guegan, Nadia Kabachi, and Khouloud Boukadi. 2021. Towards an adaptive curation services composition based on machine learning. In *2021 IEEE International Conference on Web Services (ICWS)*. 73–78. https://doi.org/10.1109/ICWS53863.2021.00022