An Approach to Build Consistent Software Architecture Diagrams Using DevOps System Descriptors

Jalves Nicacio
Université du Québec à Chicoutimi
Chicoutimi, CA
jalves.mendonca-nicacio1@uqac.ca

Fabio Petrillo
École de Technologie Supérieure - ÉTS
Montreal, CA
fabio.petrillo@etsmtl.ca

ABSTRACT

System architecture diagrams play an essential role in understanding system architecture. They encourage more active discussion among participants and make it easier to recall system details. However, system architecture diagrams often diverge from the software. As a result, they can interfere with the understanding and maintenance of the software. We propose an approach to build system architecture diagrams using DevOps system descriptors to improve the consistency of architecture diagrams. To produce our approach, we survey problems with architecture diagrams in the software industry, developing guidelines for creating architecture diagrams. Next, we produce a taxonomy for system descriptor concepts and a process to convert system descriptors into architecture diagrams. We evaluate our approach through a case study. In this case study, we defined a Docker Compose descriptor for a newsfeed system and transformed it into a system architectural diagram using the proposed approach. Our results indicate that, currently, system descriptors generally lead to consistent diagrams only to a limited extent. However, the case study’s observations indicate that the proposed approach is promising and demonstrates that system descriptors have the potential to create more consistent architectural diagrams. Further evaluation in controlled and empirical experiments is necessary to test our hypothesis in more detail.

CCS CONCEPTS

• Computer systems organization → Distributed architectures; • Software and its engineering → System description languages; Architecture description languages.

KEYWORDS

Architectural diagram consistency, System architecture, System descriptors, Modelling process, Software Systems, Software Architecture, Software Engineering

1 INTRODUCTION

Nowadays, we see a growing demand for software development that is increasingly complex and difficult to maintain. Moreover, modern software tends to be more performant and deals with quality issues such as scalability, security, maintainability, and internationalization.

For this reason, according to Fairbanks [9], software systems are arguably the largest and most complex artifacts ever built. To mitigate software’s scale and complexity issues, developers adopt processes and tools, such as software architectural modeling and, in this context, software architecture diagrams.

Software architecture diagrams are helpful for understanding and communicating about systems, as they encourage more active discussion among participants and make it easier to remember system details [14]. In this sense, software architecture diagrams play a significant role in facilitating system architecture understanding.

However, software architecture diagrams (such as UML deployment diagrams) often diverge from the software. As a result, they can adversely affect the understanding and maintenance of the software. It is a real challenge to validate the consistency of diagrams for modeling software, and several research papers have presented different approaches to verifying the consistency of models. Some recurring approaches are: (1) validation of UML models using consistency rules [1, 11], (2) algorithmic approaches [19], and (3) the use of graphs [22]. These studies validate a UML model against another UML model. For example: Checking consistency between UML sequence and state diagrams [19], between UML structural diagrams and behavioral diagrams [11], between UML component and deployment diagrams [22], and so on. Furthermore, even if the models are consistent (one with each other), there is no guarantee that these models will be consistent with the system in production. An alternative to ensure consistency of architecture diagrams with the system in production is to link them to key concepts related to continuous delivery (DevOps).

In the DevOps context, companies model and automate the high-level architecture of their systems through system descriptors. System descriptors are infrastructure as code scripts [23] to automate application configuration management and deployment [30]. For example, we can list AWS, Chef, CloudFormation, Docker Compose, Kubernetes, Puppet, and Terraform as system descriptors technologies.

Thus, software architecture and DevOps share some concerns, such as security, scalability, and system monitoring [3]. Additionally,
adoption systems descriptors proved to be a prominent approach to increasing the quality and productivity of information systems.

In light of the above, we propose an approach to build software architecture diagrams using DevOps system descriptors to improve the consistency of architectural diagrams. The main research question to be addressed is: Does creating software architecture diagrams using system descriptors improve the consistency of architecture diagrams?

To answer our research question, we report a survey on problems with architecture diagrams in the software industry, developing a set of guidelines for creating architecture diagrams. These guidelines were beneficial in the context of this work, as they helped in the qualitative assessment of the diagram produced in our approach. Practitioners can also benefit from the guidelines by understanding what qualities to expect in an architectural diagram.

Applying our guidelines, we proposed our approach and developed a case study to evaluate it. In the case study, we defined a Docker Compose descriptor for a newsfeed system and transformed it into a system architectural diagram using our approach. Finally, we compare the original and the generated diagram to evaluate the obtained results.

The contributions of the paper are the following. (i) A meta-descriptor for transforming system descriptors into architecture diagrams of distributed systems; (ii) An approach to designing consistent architecture diagrams; (iii) A set of guidelines to produce consistent architectural diagrams. Hence, the novelty of this work is the fact that it uses descriptors to generate architectural diagrams.

Our results indicate that, at the moment, system descriptors generally lead to consistent diagrams only to a limited extent. However, the case study’s observations indicate that the proposed approach is promising and demonstrates that system descriptors have the potential to create consistent architectural diagrams.

We organized the remainder of this paper as follows: Section 2 introduces the main concepts. Section 3 presents the related work. Then, section 4 presents a survey on the architectural diagrams problems. Next, the section 5 introduces our approach. We describe in Section 6 a case study on architectural diagrams generated from system descriptors. We discuss the obtained results in section 7. Finally, Section 8 concludes the article.

2 BACKGROUND

This session provides some information about the theoretical framework linked to this work and background information about the tools and technologies used to implement the proposed approach.

2.1 Architectural Diagrams

A software architecture diagram is a graphical representation that shows how the elements of a system interact in a more extensive process. In particular, they help provide an overview and context of the system.

The UML has two principal diagrams related to software architecture: component diagram and deployment diagram. In addition to UML, other efforts have been to advance software architecture, such as SysML and ArchiMate. However, according to [5], many development teams have already abandoned these notations in favor of simple "box and line diagrams" (ad hoc diagrams). Figure 1 shows an ad hoc architecture diagram of a system.

According to [14], a diagram is better than a text description of the software to promote active discussion among developers about the details of the project. There are two striking features in the diagrams: (1) Architectural diagrams help to understand the system, and (2) Architectural diagrams improve communication and collaboration [14].

Inconsistency in architectural diagrams. Litvak et al. [19] states that a consistency problem can arise because more than one diagram can describe the same aspects of the model.

To illustrate an example of inconsistency in architectural diagrams, consider the diagram in Figure 1. Analyzing this diagram, we could formulate several questions. For example, (1) how many services does this system have?; (2) where is this system deployed?; (3) how much does this diagram consistently communicate enough details to represent the same system using other languages or graphical notations, such as UML?

Thus, it is challenging to derive information about the system when certain visual elements are not present in the diagram. The idea of consistency in this paper refers to the understandability of the diagram. Thus, consistency is realized by how helpful the diagram is in understanding the system.

2.2 System Descriptors

System descriptors are scripts for automating, standardizing, and managing infrastructure in production environments. System descriptors arise in the context of Infrastructure as Code (IaC), which uses configuration scripts to specify the definition and configuration of the software infrastructure required to run a system [2].

In practice, system descriptors are artifacts that describe a system architecture. Tools such as Chef 1, Dockerfile 2, and Puppet 3 create system descriptions. Container orchestrators such as Docker-Compose and Kubernetes [8, 16] also generate system descriptors.

1Chef Infra: available at https://docs.chef.io/chef_overview/
2Dockerfile: available at https://docs.docker.com/engine/reference/builder
3Puppet: available at https://puppet.com/docs/puppet
According to [12], system descriptors allow developers to treat system infrastructure in the same way as code development: choose the right tool and implement a solution that effectively meets the requirements. System descriptors are easily replicated, versioned, and computer-executable.

While these scripts are good at telling the computer accurate information about the architecture and infrastructure of the system, diagrams do a better job of communicating data to humans [10].

### 2.2.1 System Descriptors Key Concepts

Each system descriptor in the software industry uses its own terms and concepts to describe a system. The following sections present the most important terms and concepts used by the most common system descriptors.

**Key concepts in Terraform.** The key concepts in Terraform are provider, resource, resource module, infrastructure module, composition, and data source.

**Providers** are Terraform plugins used to interact with cloud providers, SaaS providers, and other APIs. Some examples of providers are: Amazon AWS, Google Cloud Platform, and Microsoft Azure. The Terraform settings must specify which providers are required for Terraform to install and use them.

Each provider adds a set of resource types or data sources that Terraform can manage. A resource belongs to a provider, accepts arguments, creates attributes, and has a lifecycle. A resource can be created, retrieved, updated, and deleted. Some examples of resources that belong to the aws provider are `aws_vpc`, `aws_db_instance`, etc.

### 3 RELATED WORK

Concerning consistency verification in models, several research papers have presented different approaches to verifying the consistency of models. Some recurring approaches are: (1) validation of UML models using consistency rules [1, 11], (2) algorithmic approaches [19], and (3) the use of graphs [22]. However, they proposed approaches to validate a UML model against another UML model. We proposed an approach that applies a valid system descriptor to generate architectural diagrams.

In addition, other related research focused on the transformation of system descriptors into graphical representations, such as [25], [7], and [27]. One difference with our work is that we focus on improving architectural diagrams’ consistency through system descriptors.

### 4 A SURVEY ON THE ARCHITECTURAL DIAGRAMS PROBLEMS

In previous work[24], when examining system descriptors and architectural diagrams in the literature, we found a lack of research on the quality of architectural diagrams in the software industry. We also found that engineers use general-purpose notations to create ad hoc diagrams in tools such as Visio or draw.io.

Given this, we felt it was important to list guidelines that are easy to adopt and effective for improving the consistency and understandability of distributed systems architecture diagrams.

Therefore, in this session, we present a survey we conducted among software developers intending to list relevant guidelines for architectural diagram quality. The survey results are also significant for the context of this work since these guidelines lead to the quality of the architecture diagrams produced with our approach (Sec. 5).

We conducted a survey research in order to answer the following questions:

**How inconsistent are distributed systems architectural diagrams with the system they represent?** This question aims to answer the impact of the architectural diagram inconsistency on the system’s understandability and the ease of communication about the system components.

**Which aspects of architectural diagrams designed by the software engineering industry are positive and which are negative?** We investigate what practitioners comment about the main problems with architectural diagrams of distributed systems. The answer to this question helps researchers become aware of the difficulties and challenges that must be overcome when developing approaches for modeling distributed systems architecture. Also, this question helps practitioners identify the elements that act positively on the consistency and quality of architectural diagrams.

**What would be a set of best practices for designing distributed systems architectural diagrams?** The answer to this question aims to develop a set of practices that are easy to adopt and effectively improve the consistency of distributed systems architecture diagrams.

#### 4.1 Study Design

We randomly selected five architectural diagrams from various technical blogs such as LinkedIn, Netflix, Pinterest, Spotify, Diagrams.net, Pinterest.

---

1. The tool is currently called Diagrams.net
2. https://engineering.linkedin.com/blog/
3. https://netflixtechblog.com
4. https://medium.com/pinterest-engineering/
5. https://labs.spotify.com/
and Twitter\textsuperscript{9}. Figure 1 shows one of five diagrams selected for study.

For each diagram in the questionnaire, we asked three free-text questions and five multiple-choice questions phrased on a five-point Likert scale\cite{13} ranging from strongly disagree to strongly agree. We designed the questionnaire questions according to Unhelkar's model\cite{29}. We evaluated the quality of the diagrams by checking the syntax, semantic, and pragmatic (or comprehensibility) quality categories.

The scope of the survey only applies to people familiar with software development. Thus, we identified two profiles of participants in our study related to software development: professional and student. Furthermore, to ensure that participants can provide reliable answers to the questions in the questionnaire, we have defined participant exclusion criteria if they are not related to the field of software engineering.

We invited a total of 132 people. Of these, 34 responded to the questionnaire, and 33 started the survey but did not complete it. The other invited individuals did not respond to the invitation to participate in the survey.

Surveys usually have low response rates. According to Kitchenham, Budgen and Brereton\cite{15}, a score of 10% is generally considered very good. Also, according to\cite{15}, the survey must have at least 30 responses to produce any form of statistics. We have 34 participants with a response rate of 25.7%. So the number of participants made this survey possible.

The recruitment result was a sample composed mainly of mature, experienced respondents occupying a broad professional position in software engineering. The respondents were mainly from the United States and Canada, but there were also informants from Brazil. For this reason, we make the questionnaire available online in three different languages: English, French, and Portuguese.

Figure 2 presents the main results of the demographic analysis of the respondents and shows that the sample is suitable for answering the survey questions. The sample consists mainly of people between the ages of 30 and 39 (41.2%). In addition, 51.5% of respondents are undergraduates, and 39.4% of respondents are second and third-cycle graduates (24.2% Master’s and 15.2% Ph.D.).

### 4.2 Survey Results

**Diagrams consistency.** We asked participants to describe the system represented by the diagram. We then analyzed the responses to measure the similarity score between the participants’ responses and the original text. By doing so, we want to measure how far the participant’s answer is from the author’s intention. We used the cosine similarity metric to calculate the similarity score between the participants’ texts and the original text, using a scale from 0 to 1. Figure 3 shows the distribution of the similarity value in a boxplot.

Looking at the figure 3 and evaluating the degree of dispersion of the answers for each diagram, we notice that the answers for diagram 2 show a more significant variability of similarity scores. On the other hand, diagram 3 has the highest median and the lowest dispersion of similarity scores among all diagrams. Nevertheless, the responses for diagram 3 have a low similarity score.

\textsuperscript{9}https://blog.twitter.com/engineering/en_us/topics/infrastructure/
For example, the table 1 shows the keywords grouped into the category "use of shapes and arrows," as well as some excerpts from the responses that illustrate how we extracted the keywords.

When comparing the positive and negative aspects, we found that at the top of both lists, the participants emphasized the same points both positively and negatively. For example, participants mentioned that using shapes and arrows is positive, with a frequency of 20.7%. At the same time, the problems associated with shapes and arrows are the two elements most frequently mentioned as weaknesses of diagrams.

The emerging conclusion from this data is that designers generally make the most significant mistakes on the elements identified as the most important in a diagram.

Syntax, semantics, and comprehension. In the last question, we asked each participant to classify five statements according to their opinion. The statements through a scale from "strongly disagree" to "strongly agree." Thus, we used the Likert scale to interpret the data for this question.

In summary, participants tend to agree that the diagrams generally use the correct notation and that the elements are correctly labeled with the appropriate labels. However, 41% of participants disagreed or strongly disagreed that "all expected elements are present in the diagram," and 38% neither agreed nor disagreed with this statement. From the analysis of the responses, we can infer that participants lack more details to adequately interpret and understand the diagram.

5 APPROACH

We propose the use of system descriptors to improve the consistency of architectural diagrams, to build and validate them. Our hypothesis is that if an architectural diagram is generated from a valid system descriptor, then the diagram is consistent.

We assume that a descriptor file is inherently consistent for two reasons: (1) system descriptor files are written in a formal language, such as YAML (used by Docker-compose and Kubernetes pods) or Ruby (used by Chef); (2) system descriptor files are automatically processed by a finite state machine. In this way, if the diagram generated from the system descriptors represents each descriptor element, this diagram is also consistent.

There are several technologies for system descriptors that support different functions and use their modeling languages. Therefore, it is not easy to compare technologies based on their capabilities [30].

Since we intend a technology-independent approach to system description transformation, we present a taxonomy of system descriptor concepts in Section 5.1. We based the terminology used in this taxonomy on the Essential Deployment Meta-Model (EDMM) proposed by Wurster et al. [30]. Finally, in Section 5.2.1 presented the key concepts for these system descriptors.

5.1 A Taxonomy for System Descriptor Concepts

Since we intend to build architectural diagrams from a system descriptor, we must first define a set of terms representing the distributed system’s elements. For example, the UML deployment diagram uses the terms nodes for each type of hardware and artifacts for software components [26].

However, each system descriptor uses its own terms and concepts to describe a system. Moreover, different system descriptors sometimes use the same terminology but different meanings. For example, Docker Compose uses the term "services" to describe a compute resource within an application that can be scaled or replaced independently of other components [8]. Kubernetes, on the other hand, uses the terminology "service" to describe an abstraction that defines a logical set of pods and a policy for accessing them [17].

According to the meta-model proposed by Wurster et al. [30], the essential elements for the deployment of system architecture are components and relations. In addition, component and relation types specify the semantics of these elements.

Therefore, we propose a taxonomy based on observing the terminology used in Docker Compose, Kubernetes, and Terraform system descriptors. Section 2.2.1 presented the key concepts for these system descriptors.

Table 2: Taxonomy for modeling system architectures based on system descriptor terminology

| Categories            | Compose | Kubernetes | Terraform |
|-----------------------|---------|------------|-----------|
| Components            | Services | Pods, Deployment | Resource, modules |
| Components properties | Volumes, port, expose | Volumes, targetPort, port labels | Data source, Input Variables |
| Component type        | -       | -          | Resource type |
| Dependency relation   | Depends on | -          | Depends on Resources, Compositions, Infrastructure module, Resource module |
| Infrastructure nodes  | Deploy | Service, Deployment, Node | Provider, resource type |
| Artifacts             | Image, build | Image | Provider, resource type |

We classify terms according to their purpose. For example, each descriptor uses a different term to represent its minimal deployable unit: Service (in Docker Compose), Pod (in Kubernetes), and Resource (in Terraform). In this way, we have classified the key terms into six different categories, as shown in the taxonomy in Table 2.

Components. Components are deployable nodes that represent elements instantiated for system execution. They are associated with system features and business rules. Some examples of components are system modules or APIs, microservices, and databases.

Component properties. They are a set of information used to configure a component. Properties must describe either the current or desired state of a component. For example, to access files stored in locations outside the container, Docker Compose and Kubernetes use the term volume to associate the data created and used by Docker containers. Terraform, on the other hand, uses the term data source. Other examples of properties are Ports, Networks, Input Variables, and Environment Variables.

Component types. Wurster et al defines component types as an entity that specifies the semantics of a component [30]. In Kubernetes, for example, one can use key/value pairs called labels to identify a component type. In Terraform, identifying the component
We want to implement the transformation between two models: system descriptors to system architecture diagram. According to Stevens [28], model transformations aim to make software development and maintenance more efficient. The model transformation process is accomplished by automating routine aspects of the process. Once the process is automated, a model transformation can capture the notion of consistency.

Figure 4 shows the transformation model process proposed in this paper. System descriptor files are source models at the beginning of the transformation process. System descriptors can be of different types, e.g., Docker Compose, Kubernetes, and Terraform scripts.

The first step is to transform a given descriptor into a meta-descriptor (P1). Next, it transforms the meta-descriptor into a “diagram-as-code” model (P2). Finally, the rendering process maps the diagram-as-code model to the architecture diagram model.

Meta-descriptor into a diagram-as-code model. We used the taxonomy categories shown in Table 2 to create the meta-descriptor. Listings 1, 2, and 3 illustrate the P1 process. In this example, the descriptors of Kubernetes (Listing 1) and Docker Compose (Listing 2) represent the same system: a simple API whose source code is stored in the emondek/simple-api image. Therefore, in this example, the P1 process must always produce the script from Listing 3 regardless of the input descriptor.

A meta-descriptor consists of a list of components and a list of relationships between components. A component can store information about its type, properties, and artifacts. A Docker image is an example of an artifact that contains the complete stack, starting from the operating system and ending with the application-specific component.

We use the YAML-based language for the meta-descriptor, as this is a recognized standard among system description and deployment tools.

Converting the meta-descriptor into a diagram-as-code model. The P2 transformation process is responsible for converting the meta-descriptor into a diagram-as-code model. Diagram-as-code is an approach to generating diagrams through programming. This approach has sparked recent interest among software engineering tools.
practitioners. A non-exhaustive list of some web articles on the subject is given in [4, 10, 20, 21].

The transformation algorithm for the P2 process depends on the DaC model chosen. In general, DaC models describe the components and their relationships. Additional information about these models can be: the display orientation of the components (right to left, top to bottom, etc.) and the export format of the diagram.

Below, in Listing 4, we present a pseudo-algorithm that describes the main steps for the transformation between the meta-descriptor and the diagram-as-code.

Listing 4: Meta-descriptor to Diagram-as-Code algorithm

```python
1 For each component in meta-descriptor components:
   2   Generate node representation code for a specific DaC:
   3   Add properties to DaC node
   4   Add component type to DaC node
   5   Add the artifacts to the DaC node
   6   For each relation in meta-Descriptor relations:
   7   Generate the arc representation code for a specific DaC:
```

**Transformation function for system descriptors.** It is possible to describe the models transformation as bidirectional or unidirectional functions [28].

Let us assume that \( \sigma \) is a script instance of a system descriptor, and \( S \) is the set of all script instances. We also assume that \( \delta \) is an architectural diagram instance and \( D \) is the set of architectural diagrams. Hence, we define the transformation function as \( f : S \rightarrow D \), so that:

\[
 f(S) = \{ \sigma \in S \mid (\exists \delta \in D), f(\sigma) = \delta \subset D \}
\]  

is the image of \( f \).

Thus, the transformation function must always find a diagram instance that matches each system descriptor. Moreover, we know from [28] that the consistency relation is that \( \sigma \) and \( \delta \) are consistent if and only if \( \delta = f(\sigma) \).

6 CASE STUDY

In this session, we present a case study to evaluate our proposal. First, we describe a newsfeed system like Facebook or Instagram and show a high-level diagram depicting the system’s architecture. We then implement system descriptors using Docker Compose. The newsfeed application described in [31] inspired the authors.

6.1 A Newsfeed System Design

A news feed is a list of newly published content on a website. End users can receive push updates for new content on a website by subscribing to the website’s news feed. Xu [31] describes a newsfeed system in detail.

Let us assume a newsfeed system like Facebook, Instagram, or Twitter. This newsfeed system contains two streams: feed publishing and newsfeed retrieval. The feed publishing stream allows users to publish posts, and the retrieval stream summarizes friends’ posts in reverse chronological order. The newsfeed system stores the corresponding post data in the database or in the cache.

The user creates a post via the publishing service, where a news feed publishing API takes the user’s authentication details and the post data (text, photos, or videos). Then a fanout service pushes the new content to friends’ news feeds. We call fanout service the process of forwarding a post to all followers.

A modeling suggestion for this system is to design it as in Figure 5, which shows the high-level system architecture. The system components are as follows:

- **Load Balancer:** distributes the load of requests among the web servers.
- **Web server:** maintains connections with users, forwards traffic to various services, and performs user authentication.
- **Post Service:** takes care of persistence of user posts in the database and cache.
- **Fanout service:** forwards new content to the feed of friends and followers.

The Fanout service uses a communication structure based on the publisher/subscriber pattern. In this context, it is widespread for the developer to use an implemented solution to satisfy the need for message exchange between system components. Some market-ready platforms that implement this concept of producer and consumer are Apache Kafka, ActiveMQ, and Redis.

1. PlantUML is available at http://plantuml.com/
2. Mermaid is available at https://mermaid-js.github.io/mermaid/
3. Diagrams is available at https://diagrams.mingrammer.com/
6.2 A Newsfeed System Descriptor

Using the description and the architectural diagram, we created a Docker Compose descriptor. Listing 5 depicts the system components according to the Docker Compose description language. Due to space limitations, we present only part of the descriptor in this section. Readers interested in more details can find the full version in our replication package.

Note that the nginx service (see line 2 in Listing 5) has the role of load balancing according to the system description, although the Docker Compose descriptor does not contain this information. This issue affects the creation of the meta-descriptor, as it is not possible to assign missing information.

Table 5 also summarizes all Docker services created for the newsfeed system and some important aspects of these services. So we see that Docker Composer has eight services specified. Moreover, some of these services are linked via the depends_on property, creating a dependency relationship between them.

```bash
services:
  nginx:
    image: nginx:latest
    restart: always
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf:ro
    depends_on:
      - webserver
```

Listing 5: Docker Compose newsfeed system descriptor

6.3 Transformation P1

If we use the system descriptor in Listing 5 as input to the algorithm in Algorithm 1 we get the meta-descriptor in Listing 6. This task corresponds to process P1 in our approach.

We show only a fragment of the code generated for the meta-descriptor in Listing 6. Comparing the nginx service from Listing 5 with the nginx component of the meta-descriptor, we see that the algorithm capture the essential elements for generating the architecture diagram.

```bash
components:
  nginx
  name: nginx
  properties:
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf:ro
  artifacts:
    name: nginx:latest
  relations:
    - out: nginx
      in: webserver
```

Listing 6: Newsfeed System meta-descriptor

6.4 Transformation P2 and Rendering

We have chosen to use an automated tool for rendering the diagram. For this reason, the P2 process of our approach generates a diagram-as-code artifact that corresponds to the DSL language of the Structurizr tool. Listing 7 presents a fragment of the Diagram-as-code model.

```bash
workspace {
  model {
    softwareSystem = softwareSystem "Newsfeed System" {
      nginx = container "nginx" "nginx:latest"{
        webserver = container "Web Server" "Replica: 3" "httpd:latest" "Replicas: 3"{
          db = container "db" "database" "postgres:14.1-alpine"{
            cache = container "cache" "redis:6.2-alpine"{
              worker = container "worker" "app-image:latest" "redis" "replicas: 3"
            }
          }
        }
      }
    }
  }
}
```

Listing 7: Newsfeed System Diagram-as-code model
7 DISCUSSION

We compared the architectural diagram of Xu [31] (Figure 5) with the architectural diagrams rendered in our case study (Figure 6), and we discuss our observation as follows.

First, we note that the architecture diagram contains all the elements of the system descriptor. Since our transformation function (consisting of P1, P2, and rendering processes) has succeeded in generating a diagram δ from a system descriptor σ, such that \( \delta = f(\sigma) \), then we can say that the rendered architectural diagram is consistent.

**Observation 1:** The rendered architectural diagram accurately describes the system descriptor elements and their relationships.

Analyzing the diagram in terms of the guidelines presented in Section 4.2, we can see that all guidelines have been met, except for guideline number 4, since this type of information is missing from the system description. In fact, some elements of the original architecture diagram in Figure 5 do not appear in the rendered diagram, such as user elements (web browser and mobile app) or DNS elements. This is because these elements are external nodes that interact with the system and, for this reason, are not included in the system descriptors, regardless of the technology used.

However, we believe that external elements play an essential role in understanding systems because they relate the system to external entities, provide an overview of how the system interacts with external elements, and contribute to greater consistency in the architectural diagram.

**Observation 2:** External nodes play an important role in the diagram comprehension and should be rendered.

Finally, we found it challenging to map the system descriptor to the meta-descriptor because it was difficult to identify the essential elements of the meta-descriptor (components, component types, and their relationships) in the system descriptor. For example, Kubernetes does not inherently support the use of a term like `depends_on` in its descriptor, which makes it difficult to map relationships between components. Similarly, in Docker Compose and Kubernetes, it was not possible to identify the existence of a specific term that informs about the type of component.

A solution to this problem in Kubernetes is for the developer to create custom terms for this purpose. In Docker Compose, on the other hand, a solution could be to use another term, such as `container_name`, to identify the type of component. Nevertheless, these solutions depend on the quality and details of the system descriptor document written by the developer and would make it difficult to automate the transformation process.

**Observation 3:** The difficulties encountered in mapping elements from system descriptors to meta-descriptors make the P1 process more challenging and complex to automate.

7.1 The Consistency of Architectural Diagrams

This section gives answers to research question: *Does building software architecture diagrams using system descriptors improve the consistency of architecture diagrams?*

In light of the preceding observations, we have found that system descriptors generally lead to consistent diagrams only to a limited extent. However, based on the evidence shown in this paper, we observe the potential of system descriptors to create architectural diagrams.

We suggest adding annotations in that system descriptors to avoid the difficulties encountered in mapping descriptors. These annotations would highlight missing or difficult-to-assign information from the descriptors (e.g., component type and dependency relationship). Developers should insert the annotations as comments in the code to avoid disturbing the processing of the system descriptors. Finally, the P1 process of our approach would read these annotations and use the information to complete the mapping of the components and their relationships.

We justify these adaptations’ costs by the benefits of this approach to creating architectural diagrams. The advantages of creating architectural diagrams from descriptors are:

1. the architecture diagrams can represent all descriptor elements.
2. the architecture diagrams remain synchronized with the evolution of the descriptors and consequently of the system.
3. the architecture diagrams avoid inconsistency errors due to system evolution, as diagrams synchronize with system descriptors.

7.2 Threats to Validity

This section identifies and discusses threats that could affect the results’ validity. We identify the following threats.

Regarding the external validity, some aspects could affect the general applicability of our results. Our case study refers to modeling a single system using only one system descriptor provider applying the approach. For this reason, we cannot generalize the results. Nevertheless, we believe the results are valid as they provide relevant insights into the potential of descriptors to build architectural diagrams.
For example, the architectural diagram rendered in the case study could clearly represent all system descriptor elements. In addition, Wurstner showed that it is possible to map several other system descriptor providers to the EDMM meta-model, such as Puppet, Chef, Ansible, Terraform, OpenStack, SaltStack, AWS, Azure, Kubernetes, and Docker Compose [30].

Since we are using the EDMM meta-model to develop the taxonomy of system descriptor terms, we believe it is possible to generate a meta-descriptor from each of the above descriptors. We intend to explore these issues in more detail in future work.

8 CONCLUSION

In this paper, we present our approach to applying system descriptor concepts and a function for converting system descriptor elements into architectural diagrams. Further, we presented a case study evaluating our approach.

Our case study shows that the rendered architecture diagram accurately describes the system descriptor elements and their relationships. However, we found that some external system elements present in the original architecture diagram, such as user elements, do not appear in the rendered diagram because the descriptor does not include these elements. We believe that external nodes play an essential role in understanding the diagram and, therefore, should be rendered. In addition, we found that the difficulty in mapping elements from system descriptors to meta-descriptors makes the P1 process more difficult and complex to automate. Additionally, some details in the descriptors, such as service ports, could be incorporated into architectural diagrams as an additional information layer. Also, other details, such as passwords, were not mapped in the generated architectural diagram. Such information describes details that are out of the scope of architectural diagrams. Unmapped elements of a system descriptor can be specified as a subset of the domain of that mapping.

Based on these insights, we have found that system descriptors generally lead to consistent diagrams only to a limited extent. However, the case study’s observations indicate that the proposed approach is promising and demonstrates that system descriptors have the potential to create more consistent architectural diagrams.

Our case study only uses a single system descriptor provider: Docker Compose. In future work, we intend to carry out studies with other descriptor providers. Also, we intend to implement a tool to evaluate our hypotheses more deeply in a controlled experiment.

REFERENCES

[1] S. O. B. Amor, M. Ali, and F. Gargouri. 2011. Verification of the consistency between use case and activity diagrams: A step towards validation of user requirements. In ICSES 2011 - Proceedings of the 15th International Conference on Enterprise Information Systems, Vol. 3 ISAS. 396–399.

[2] Matej Artic, Tadej Borovšak, Elisabetta Di Nitto, Michele Guerriero, and Damian Andrew Tamburri. 2017. DevOps: Introducing Infrastructure-as-Code. In 2017 IEEE/ACMI 39th International Conference on Software Engineering Companion (ICSE-C). IEEE.

[3] L. Bass, I.M. Weber, and L. Zhu. 2015. DevOps: A Software Architect’s Perspective (1st ed.). Addison-Wesley.

[4] Simon Brown. 2020. Modelling software architecture with PlantUML. https://dev.to/simonbrown/modelling-software-architecture-with-plantuml-56c.

[5] Simon Brown. 2021. The C4 model for visualising software architecture: Context, Containers, Components, and Code. https://c4model.com/. Accessed: 2021-09-21.

[6] Simon Brown. 2022. Diagrams as code 2.0. https://structurizr.org/.

[7] Fabio Burco, Marino Miculan, and Marco Peressotti. 2020. Towards a Formal Model for Composable Container Systems. In Proceedings of the 35th Annual ACM Symposium on Applied Computing (Breno, Brazil Republic) (SAC’20). Association for Computing Machinery, New York, NY, USA, 173–173.

[8] Docker Docs. 2019. Overview of Docker Compose. https://docs.docker.com/compose.

[9] G. Fairbanks and David Garlan. 2010. Just Enough Software Architecture: A Risk-Driven Approach. Marshall & Brainerd.

[10] Hans Kristian Flaattem. 2020. Diagram as Code. https://open.evry.blog/2020/01/05/diagram-as-code.html.

[11] I. Ha and B. Kang. 2003. Meta-Validation of UML Structural Diagrams and Behavioral Diagrams with Consistency Rules. In IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing - Proceedings, Vol. II. 679–683.

[12] Michael Hüttermann. 2012. Infrastructure as Code. Apress, Berkeley, CA, 135–156.

[13] Susan Jamieson. 2005. Likert Scales: How to (ab) Use Them. Medical education 38 (01 2005), 1217–8.

[14] Rodi Jolak, Maxime Savary-Leblance, Manuela Dalibor, Andreas Wortmann, Regina Hebig, Juraj Vinceur, Ivan Polasek, Xavier Le Pallec, Sebastien Gerard, and Michel R.V. Chaudron. 2020. Software Engineering Whispers: The Effect of Textual Vs. Graphical Software Design Descriptions on Software Design Communication. Empirical Software Engineering (June 2020).

[15] Barbaraa Ann Kitchinham, David Buddgen, and Pearl Brereton. 2015. Evidence-Based Software Engineering and Systematic Reviews. Chapman & Hall/CRC.

[16] Kubernetes. 2021. Pod Overview. https://kubernetes.io/docs/concepts/overview/

[17] Kubernetes. 2022. Service. https://kubernetes.io/docs/concepts/services-networking/service.

[18] HashiCorp Learn. 2022. Terraform Language Documentation. https://www.terraform.io/language.

[19] B. Litvak, S. Tyszberowicz, and A. Yehudai. 2003. Behavioral consistency validation of UML diagrams. In Proceedings - 1st International Conference on Software Engineering and Formal Methods, SEFM 2003.

[20] Raoul Meyer. 2019. Infrastructure diagrams as code. https://devto/raoulmeyer/infrastructure-diagrams-as-code-3fj.

[21] Mingrammer. 2020. Introducing Diagrams: Diagram as Code. https://medium.com/better-programming/diagrams-diagram-as-code-5f0ec2224d6.

[22] R. G. Mohammad and A. A. Barforough. 2014. Enforcing component dependency in UML deployment diagram for cloud applications. In 7th International Symposium on Telecommunications (IST 2014). 412–417.

[23] Kief Morris. 2016. Infrastructure as code: managing servers in the cloud. O‘Reilly Media, Sebastopol, CA. https://cids.cern.ch/record/2157549.

[24] Jalves Nicacio and Fabio Petrillo. 2021. Towards improving architectural diagram consistency using system descriptors. In 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC). 401–405.

[25] Bruno Padade, João Pedro Dias, and Filipe F. Correia. 2020. An Empirical Study on Visual Programming Docker Compose Configurations. In Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings (Virtual Event, Canada) (MODELS '20). Association for Computing Machinery, New York, NY, USA, Article 60, 10 pages.

[26] James Rumbaugh, Ivar Jacobson, and Grady Booch. 2004. The Uniﬁed Software Development Process. 2nd Edition. Prentice Hall.

[27] Julio Sandobalin, Emilio Insfran, and Silvia Abrahão. 2019. ARGON: A Model for Composable Container Systems. In Proceedings of the 13th International Conference on Analytic Models of Computing (ICSMC). 738–742.

[28] Perdita Stevens. 2008. A Landscape of Bidirectional Model Transformations. Springer Berlin Heidelberg, Berlin, Heidelberg, 408–424.

[29] B. Unhelkar. 2005. Evaluation of UML Diagram Consistency in UML deployment diagram for cloud applications. In 7th International Symposium on Telecommunications (IST 2014). 412–417.

[30] Michael Wurstner, Uwe Breitenbuecher, Antonios Brogi, Gheareb Falazi, Lukas Harzenetter, Frank Leymann, Jacopo Soldani, and Vladimir Yussupov. 2019. The EDMM Modeling and Transformation System. Springer-Verlag, Berlin, Heidelberg, 5 pages.

[31] A. Xu. 2020. System Design Interview: An Insider’s Guide. Independently Published. https://books.google.ca/books?id=TZxWmQ6ACAAJ

[32] Xiang Zhang and Chuck Jerian. 2020. How we improved latency through project in Espresso. https://medium.com/pinterest-engineering/how-ads-indexing-works-at-pinterest-99857f68289. Accessed: 2020-06-29.