The Weakest Link: Revealing and Modeling the Architectural Patterns of Microservice Applications

Vladimir Podolskiy
v.podolskiy@tum.de
Chair of Computer Architecture & Parallel Systems
Technical University of Munich
Germany

Maria Patrou
maria.patrou@unb.ca
Faculty of Computer Science
University of New Brunswick
Canada

Panos Patros
panos.patros@waikato.ac.nz
Department of Software Engineering
University of Waikato
New Zealand

Michael Gerndt
gerndt@in.tum.de
Chair of Computer Architecture & Parallel Systems
Technical University of Munich
Germany

Kenneth B. Kent
ken@unb.ca
Faculty of Computer Science
University of New Brunswick
Canada

ABSTRACT

Cloud microservice applications comprise interconnected services packed into containers. Such applications generate complex communication patterns among their microservices. Studying such patterns can support assuring various quality attributes, such as autoscaling for satisfying performance, availability and scalability, or targeted penetration testing for satisfying security and correctness. We study the structure of containerized microservice applications via providing the methodology and the results of a structural graph-based analysis of 103 Docker Compose deployment files from open-sourced Github repositories. Our findings indicate the dominance of a power-law distribution of microservice interconnections. Further analysis highlights the suitability of the Barabási-Albert model for generating large random graphs that model the architecture of real microservice applications. The exhibited structures and their usage for engineering microservice applications are discussed.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; • **Software and its engineering → Extra-functional properties**.

KEYWORDS

cloud-native application, microservice, software vulnerability, application topology

ACM Reference Format:
Vladimir Podolskiy, Maria Patrou, Panos Patros, Michael Gerndt, and Kenneth B. Kent. 2020. The Weakest Link: Revealing and Modeling the Architectural Patterns of Microservice Applications. In *Proceedings of 30th International Conference on Computer Science and Software Engineering (CASCON'20).* ACM, New York, NY, USA, 10 pages.

CASCON'20, November 10–13, 2020, Toronto, Canada

1 INTRODUCTION

Cloud-native applications comprise containerized microservices, each implementing a narrow part of the application's functionality to enable fine-grain elasticity. However, finding the right number of containerized microservice instances to guarantee quality of service, reduce resource consumption and identify bottlenecks is not trivial: communication between microservices can happen in a number of ways depending primarily on the application's topology.

Analyzing the structure of real microservice applications unveils chains of microservices (in a producer-consumer relationship) that utilize various communication protocols with as many as 100-300 services [7]. When scaling a particular logical service in such a chain, one may face the necessity of cascading capacity changes for the downstream services to avoid such services becoming a new bottleneck [19]. Knowing the topology of a microservice application could help identify such a bottleneck service, *the weakest link*, in advance. This allows predictive scaling that can dynamically meet demand [26] and protect against malicious entities exploiting *a weakest link vulnerability*, which can happen via a targeted denial of service attack affecting the availability of the cloud application.

Besides scaling and security, knowing a microservice topology can assist in assuring other software quality attributes. For example, realistic benchmarks can be created using the service topology as a generic template. Also, deployment based on the weakest link-services to assist the weakest one and determining the application capacity can lead to better performance.

The lack of publicly available industry-scale microservice applications precluded the research of this type of applications [7]. The study of public code repositories allows us to overcome this challenge to an extent as individuals and companies tend to open source their production code or community projects. The strong positive correlation between the number of employees and number of services supported in an application [7] allows us to assume that it is more likely to find an application encompassing a high number of services and with a more complex topology in the public repository of a large company, such as Google or Uber, than in the repository of a recent startup or an individual.

Studying the structure of open source microservice applications can disclose common topological patterns. Generating versatile real-like application structures from these patterns can further be used to assemble microservice applications of similar structure but with a larger size to enable practice-relevant research or realistic stress-testing for such applications. We focus on the architectural patterns of microservice applications contributing in:

- Performing an empirical study of the structure of 103 microservice applications available on Github.
- Modeling the structure of the over-represented microservice application type with a power-law distribution of vertices' degree using random graph models, which we evaluate.
- Outlining an overall methodology for performing such empirical studies, including the identification of an appropriate random graph generation model and tuning its parameters.

The paper is organized as follows: background and related work in Sections 2 and 3; architectural pattern inspection of the applications in Section 4; modeling of the structure of applications with power-law service degree distribution in Section 5, observations in Section 6 and conclusion and future work in Section 7.

2 BACKGROUND

A microservice implements a limited functionality, is independently deployable and often communicates with other microservices via the network. OS-level virtualization with containers allows one to implement microservices easily—the software developer needs to add the necessary libraries and the software to the container image, which can be used to deploy multiple containers implementing the same function. In a microservice application that follows a Service-Oriented Architecture (SOA) software design style, the communication between the microservices is usually done via API calls over the network—this supports loosely coupled applications and allows fine-grained application elasticity.

Microservice applications tend to serve users' requests, a pattern commonly used in web-shops and online-portals, because this architecture addresses multiple requirements: the response time of a microservice application deployed on the cloud can be relatively short and predictable by scaling individual microservices; high availability is ensured by negligible microservice deployment times; and there are multiple orchestration tools available for microservice applications, which make management and autoscaling easy tasks (e.g., Docker Swarm and Kubernetes) [16].

The application deployment in Docker Swarm requires the use of a Compose file [18] in YAML [2] standard, which describes the components of the application and their interconnections at a software architecture and deployment level. Docker Swarm initializes the cluster with the container-services described in the YAML file. The file includes configuration settings for each service that resides in a container, including the container image, which has the executable code, and its dependencies with other containers that affect the order of starting and stopping the services. Furthermore, information on the cluster's networking for intercommunication and reachability among containers and their data storage is defined.

While in Docker Swarm, applications are organized into containers, Kubernetes leverages Pods. Each Pod has one or multiple containers, while groups of pods are deployed to create an application on a (cloud) cluster [27].

2.1 Graph Theory Essentials

A graph is a discrete mathematical abstraction that encompasses a set of objects (*vertices* of a graph) and a set of relations between these objects (*edges* of a graph). If the set of vertices is denoted by V and the set of edges is denoted by E with an edge between i^{th} and j^{th} vertices being e_{ij}, then a graph can be denoted as an ordered pair $G = (V, E)$. Graphs are usually depicted with circles being the vertices and lines being the edges; if a graph is directed (the order of vertices in edge matters), arrows are used instead of simple lines. A graph can be quantified by parameters. The most basic quantification is through the number of vertices, $|V|$, and the number of edges, $|E|$. In addition, each vertex could be quantified by the number of edges that connect to it; this parameter is called the *degree* of a vertex, $deg(v)$. In a directed graph one can further divide the notion of degree into outdegree, i.e.,the number of edges that start at this vertex, and indegree, i.e., the number of edges that end at this vertex. The degree sum for the undirected graph could be computed as $\sum_{v \in V} deg(v) = 2|E|$. The degree of a vertex is a primary characteristic of structural patterns in the graph as it can be used to describe the connectivity of a particular part of a graph by relating vertices to edges in a quantifiable way. Thus, graph theory is used in the paper as a formalism to analyze the structure of microservice applications.

2.2 Network Theory Essentials

Network theory emerged to address the complexity and vulnerability of real-world structures like power grids or the Internet [5]. In essence, modelling real a structures, a network is a graph with labelled vertices and/or edges.

It is often necessary to understand which nodes in the network are more important than the others. The importance could be denoted differently, but the most common way is to associate the number of connections with a node's importance. The identification of such nodes is addressed by *centrality indices* that are computed differently [9]. Degree centrality is one of the simplest centrality measures and is defined as the number of links incident upon a node, thus the degree centrality of a vertex v is $C_D(v) = deg(v)$; it characterizes the immediate importance of the node.

Degree distribution is a probability distribution of degrees in the network used to describe the whole network. Degree distribution shows how often nodes with a particular degree are encountered—different degree distributions correspond to different structures. For example, if the degree distribution has a long tail for higher degrees, then the network contains only a few nodes of high importance, i.e., numerous connections with other nodes. This fact can have significant implications in such cases as developing a network structure that is resilient to cyber attacks. Hence, certain structural properties of the network can be conveyed with the degree distribution.

The degree distribution of a network can be approximated by a formula; this allows the in-depth study of the network's properties. For example, one of the most common types of networks is a scale-free network with the probability distribution described roughly by $P(d) \propto d^{-\alpha}$, where the fraction of nodes with d connections is

defined as $P(d)$ and drops exponentially with the growth of the degree (α is usually between 2 and 3). Various models exist to describe the properties of the networks and to generate new ones. In particular, scale-free networks are best described by the Barabási-Albert (BA) model that uses a preferential attachment method to generate networks with a power-like degree distribution [4].

We employ degree distribution on the microservice connectivity as described by the configuration files. The metric exposes the connectivity across the microservices, thus revealing the application's structure model. We conduct an analysis of the microservice applications that allows us to use random graph models to generate networks with realistic microservice structural properties.

3 RELATED WORK

Despite the absence of work devoted to the study of the structural aspects of microservice applications, the importance of such research is recognized in the literature [14].

Contributions to the study of application's structure were made for conventional multi-tier application architectures, such as ones with a front-end, an application service and a database. The necessity to incorporate such knowledge to identify application bottlenecks was recognized by Malkowski et al. as the result of experimental studies of N-tier applications using the RUBiS and RUBBoS benchmarks [24]. Wang et al. approached the challenge of detecting the transient bottlenecks in multi-tier applications that contribute to the latency long-tail problem in clouds via elaborate load-throughput analysis on multiple tiers of application [32, 33]. Liu et al. applied queuing network theory-based application modeling to wide-spread 3-tier web-applications to derive accurate predictions for response time and throughput [23]. Workload scaling as a method to scale multi-tier cloud applications via replicating the processing of the same request and sending the results of the fastest VM to the user was proposed by Pérez at al. [28]; the same work marks application topology and tier-specific workload scaling models as a research challenge. sPARE is the first known partial replication system that takes into account the structure of a multi-tier application to coordinate the replication levels on all tiers [6].

Similar to us, Márquez et al. performed an empirical study on scalability aspects of microservice-based applications by investigating 30 open-sourced projects. They analyzed three types of configuration files found in the projects: YAML files for Docker Compose, POM files for Apache Maven and Gradle files (build.gradle). Their main focus was to answer research questions towards scalability using their pattern language that focuses on scalability dimensions they have previously identified. Their goal is to identify the frameworks that meet the scalability dimensions and provide recommendations on microservice architecture [25].

In contrast, our study focuses on applying graph theory to microservice-based applications' structures. Every service, which is defined in the Compose file, is treated as a node and keywords that reveal their dependencies are used to extract the connections among services. We identify and generate models that best fit the structures and discuss their potential usage in software engineering. Any observations on software qualities, such as scalability, security etc., are made based on the architecture of the service interconnection and can be used to improve software quality. Finally, unlike other empirical studies who analyze source code, such as [11], our focus is entirely on the interconnections at the software architecture/component level.

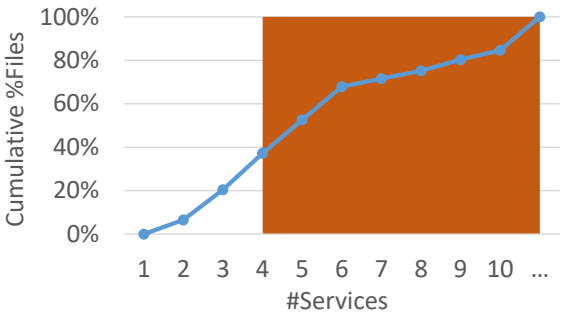

Figure 1: Cumulative distribution of configuration files per number of services they declare. The shaded area designates the files having four or more services for further analysis.

4 ARCHITECTURAL PATTERN INSPECTION

Various public Github repositories of IT companies, organizations and individuals were explored manually to obtain their configuration YAML files. The files were processed to reveal the type of services and how they tend to be interconnected. A statistical analysis with graph and network metrics was applied to find how common distributions model real-world application structures.

4.1 Dataset

Although following a manual data collection is a limitation of the study, it was adopted since the automatic exploration of Github repositories' excerpts available on Kaggle[1] would result in meaningless sample pet-projects polluting the results and thus, biasing the resulting structural models. The exploration resulted in a collection of 137 Docker Compose configuration files taken from 107 Github repositories, which we made available [1]. The collected configuration files represent a variety of applications, including web-shops and web-portals, cloud platforms for IoT, technology stacks, etc. Their version distribution was as follows: 18 files were version 1.0; 66 were version 2.x; and 53 were version 3.x.

These YAML files define the start up sequence of microservices via special keywords, such as *depends_on*, *links*, or *external_links*[2]. We used this formal specification to create service-dependency graphs. However, to ensure our dataset contained complex-enough points, it was decided to exclude those that had three or fewer connected microservices (Figure 1).

After the above filtering was processed, 103 Compose configuration files were selected and analyzed for their microservice topology. Based on these configuration files, the results show that at least 26% of the applications contain more than eight microservices. For each of the 826 services defined, we extracted the number of ports exposed to other services and to outside clients, the number of persistent volumes, the number of services a service depends on and the number of services that are depended upon a service.

[1] https://www.kaggle.com/github/github-repos
[2] https://docs.docker.com/compose/compose-file/#links

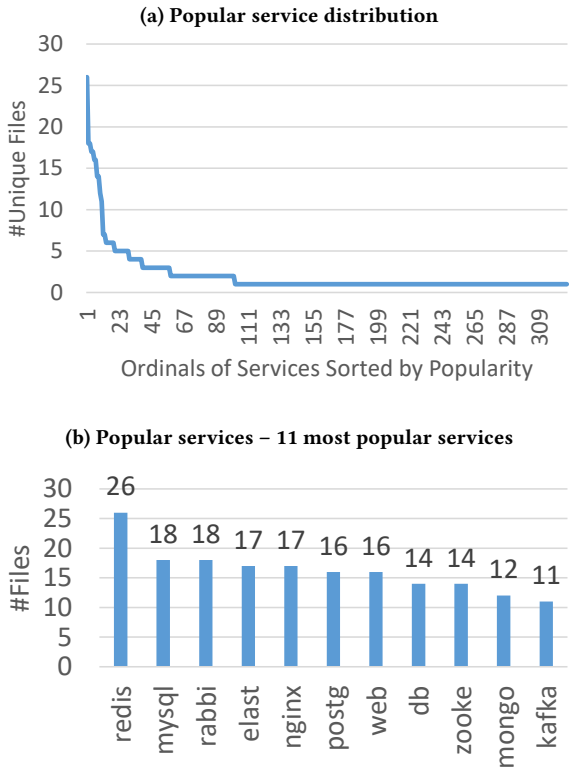

Figure 2: Service popularity in the filtered dataset

General observations were made on the filtered service data using the Pearson Coefficient to show the relationship among the metrics (as values move away from zero, the statistical relationship among the metrics is stronger):

- No large correlations were measured among the extracted metrics. A coefficient of -0.19 was recorded between the numbers of depending and depended services, which indicates the presence of leaves and roots in the tree-structure of an application; and a coefficient of 0.17 for the number of volumes and dependent services, which indicates that persistence-related functionality was less frequent on leafs.
- No trend was observed for an increased number of microservices being used as versions progressed. The correlation between number of services and file version was 0.06.
- Certain services were more popular than others in a way that resembles a power law distribution (Figure 2). In particular, popular services used in at least ten different files revealing various databases, cloud elasticity services and load balancers—not surprising, given the types of applications commonly deployed on the cloud.

4.2 Microservice Degree Distribution

Next, an adjacency matrix representation of the underlying directed graph was produced for each of the deployment files of the dataset. The matrices were analyzed to identify patterns in the structure of the microservice applications using the observed degree metric, i.e., the number of services that are connected to a service.

4.2.1 Degree Distributions. Visual inspection identified three prevalent degree distribution types in the dataset:

- **Uniform distribution**. The number of vertices N for a degree d in a range $[a, b]$ is *const* and 0 otherwise.
- **Power law (Pareto) distribution**. The number of vertices N for a degree d is $N \propto d^{-\alpha}$.
- **Normal distribution**. The number of vertices N for a degree d is $N \propto e^{-\frac{(d-\mu)^2}{2\sigma^2}}$.

Furthermore, an automated machine learning-based approach, agglomerative clustering, that leveraged hierarchical clustering, also identified three distinct clusters in our dataset, confirming our visual inspection findings. Each application was assigned a vector $v = (p_0, p_1, ..., p_{24})$ that represents the probability p_i of a service being connected by i services in a specific application, where 24 is the maximum observed number of services that a service is connected by. Each connection is represented as a directed line from one service to the others that depends, as indicated by the Compose file's keywords: *links*, *external_links* and *depends_on*. As an example, consider Figure 4C, which shows the graph of a batch scheduling system by Yelp with four services and three connections. The calculated service dependence probability vector is: 0.25,0.75,0,0,..,0, where as 25% of the services (one) have no incoming line to them and 75% (three) have one.

The service dependence probability vectors were averaged, clustered pairwise and recursively based on the smallest Euclidean distance among their probabilities. The dendrogram of the results is displayed in Figure 3 and shows the three clusters that the samples were automatically grouped into. When averaging out all members of each of the clusters, the aggregate distributions (which are omitted for brevity) appear to be primarily following the power law with some other distribution added on top.

However, every cluster shows different properties. Cluster 1 has more than half (57%) of its services with one service to be depended upon and 28% zero. Thus, the majority of the services were acting as leaf-services and less than a third as roots. Cluster 2 has 71% of the application's services with no dependencies, indicating minimum dependency among the majority of the services, while the dependencies should be concentrated to a few services. Cluster 3 is the most representative in the dataset, comprising 44% of the applications. It indicates that 40% of the services had zero dependencies, 13% and 27% had one and two, respectively. The cluster shows more services with zero dependencies that Cluster 2, revealing fewer independent services and more services with less than two dependencies. All three clusters show that dependencies among services do not exceed two connections for most services: finding a service that depends on more than two services is rare.

4.2.2 Degree Distribution Methodology for Small Graphs. With a maximum of 24 vertices in the largest microservice application, determining the form of the degree distributions with statistical tests can be inaccurate [20]. A graph may be attributed to several distribution types. To improve the quality of such tests, we devised an appropriate testing technique.

The proposed approach combines *conventional statistical distribution tests* with fallback *heuristics*. Preliminary tests on randomly generated distributions showed high inaccuracy of statistical tests

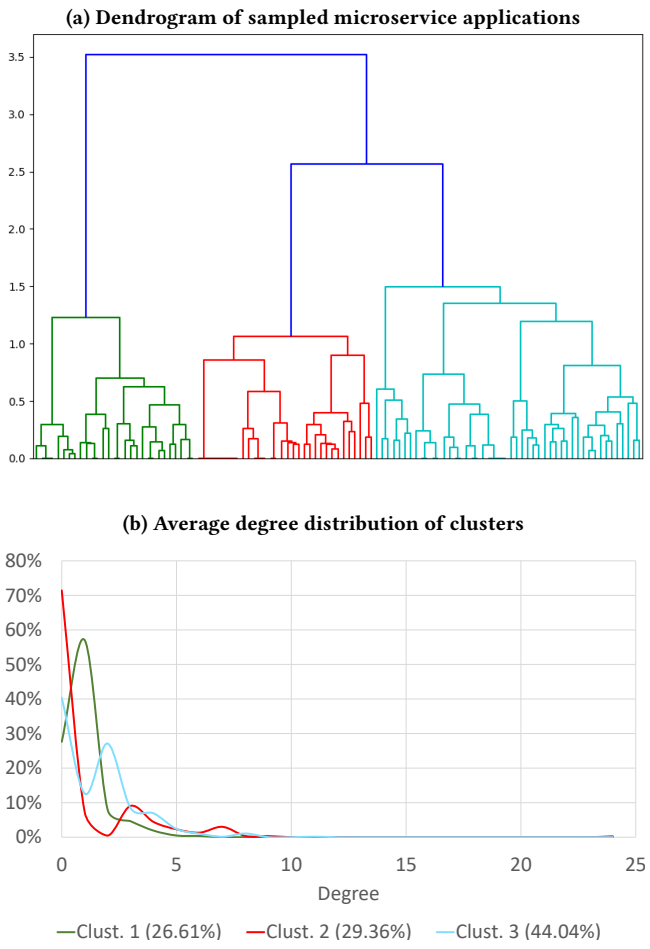

Figure 3: Unsupervised learning of clusters of distributions in the dataset.

for a number of samples less than six; hence we applied fallback heuristics when graphs had fewer than six vertices or when the corresponding statistical test could not be applied to the degree distribution. Although the merger of the statistics and heuristics-based analysis approaches is limited compared to the pure statistics, statistics offers relatively few methods available for the small population sizes. Omitting the small applications (between 3 and 6 services) from the consideration would have added a significant flaw to the research since there exist industry applications of such "small" sizes, e.g. at companies where IT plays only the support role for the operations [7]. The designed heuristics are as follows:

Uniform distribution heuristic. A small number of distinct degrees in graphs makes the direct application of uniform distribution tests impractical. However, it is possible to transform the data such that statistical testing would provide meaningful results. First, the initial degree distribution is transformed into a histogram. Following, the Pearson's chi-squared test is applied to test the degree distribution based on a Monte Carlo test with 500 replicates [17]. The value of 500 replicates was determined by conducting multiple tests on randomly generated distributions. The fallback heuristic for uniform distribution checks the single outcome not covered by the statistical test: when all vertices have the same degree.

Power law distribution heuristic. The Kolmogorov-Smirnov test was used to determine if the degree distribution of a graph is close to a power law (Pareto) distribution. Computed parameters of power law distribution allow us to determine if the fallback test should be invoked. Usually, it is necessary for borderline graphs with 6–7 vertices. The fallback heuristic computes the mean degree and checks if the number of vertices with a degree lower than the computed mean is higher than the number of vertices with a degree higher than the mean:

$$|\{v_i | d \leq \mu\}| - |\{v_j | d > \mu\}| > T$$

Based on the threshold T for such a comparison, more or fewer cases can be classified as following the power law; the threshold values 1 or 2 were good for the collected dataset.

Normal distribution heuristic. To determine if the degree distribution of a graph follows a normal distribution, the Shapiro-Wilk test of normality [29] was used. This test was shown to be more powerful when testing for normality in comparison to Kolmogorov-Smirnov [31]. Its associated fallback heuristic checks 1) if the most frequent degree in a graph d_f is between the minimal (d_m) and maximal (d_M) degrees, and 2) if the number of vertices with degrees higher than the most frequent degree and the number of vertices with degrees lower than the most frequent degree are almost equal (discrepancy by a threshold $T = 1$ was allowed):

$$\left(d_m < d_f < d_M\right) \wedge \left(|\{v_i | d \leq d_f\}| - |\{v_j | d > d_f\}| \leq T\right)$$

4.3 Service Degree Distribution Analysis

We compared the known distributions: power-law, uniform and normal with the application topologies using the statistical tests and the heuristics described above. Table 1 shows the applications that fit in the corresponding distribution type under the graph-based threshold parameters. To account for the limitations of the statistical analysis with fallback heuristics, we adapted the distribution types names accordingly. Both absolute numbers and percentages in dataset are reported. The *Total* column presents the applications that have the distribution type, while the *Pure* column shows the applications that fit only in the underlying distribution type.

Microservice applications with the power law degree distribution of the underlying structure graph prevail. The applications with such a degree distribution cover around 87% of the whole data set with a loose threshold of 1 for the fallback heuristic and around 78% with a tighter threshold of 2. The uniform and normal distribution cases amount to only around 42% and 19% of cases correspondingly. Considering only the cases that were associated with a single distribution type, a similar picture of power law distribution emerges, being the most frequent with around 47% of all the cases, and followed by the uniform distribution with around a 30%-wide gap. For the small number of unique degrees, the uniform degree distribution might be overrepresented. Hence, for the examined dataset, the dominance of the power law-like distribution becomes even more apparent. Samples of the discussed graphs can be found in Figure 4.

Distr. Type	Threshold = 1[a]		Threshold = 2[a]	
	Total[b]	*Pure*[c]	*Total*[b]	*Pure*[c]
Skewed	90 (87.4%)	48 (46.6%)	80 (77.7%)	42 (40.8%)
Near-uniform	43 (41.8%)	11 (10.7%)	43 (41.7%)	14 (13.6%)
Central	20 (19.4%)	0 (0.0%)	20 (19.4%)	0 (0.0%)
Other	- (-%)	2 (1.9%)	- (-%)	8 (7.8%)

[a]Threshold is set for the fallback test.
[b]Positive outcomes for other types are possible.
[c]Only negative outcomes for other distribution types.

Table 1: Degree Distribution types

Table 1 shows several distributions that are different from those tested. The thresholds increase from one to two for the power-law heuristic test yields an increase in the number of unclassified cases by six, which might be hybrids between the skewed and some other types . The two other cases should be quite different from the power law distribution. Indeed, these two examples show the prevalence of vertices of a higher degree in comparison to vertices of a lower degree; this type of distribution could be described as $N \propto e^{d}$.

The main outcome of the analysis is that most applications have a structure of a *scale-free network* [3]. The skewed degree distribution with a long tail implies a presence of services that have significantly more connections than others; there are at least several types of such microservices, e.g., PostgreSQL, Zookeeper, RabbitMQ and Elasticsearch. This is not surprising as these microservices implement common functions, such as logging, configuration management, message brokering and data storage. This also means that most microservice applications *tend to form bottlenecks* and *are susceptible to targeted attacks.*

5 ARCHITECTURAL PATTERNS MODELING

Our dataset provides hints on how cloud-native applications tend to be structured. Understanding these tendencies can result in models that capture structural properties of real-world applications for further structure-driven capacity balancing research. To evaluate what types of models better fit our data, we use several models that can generate random graphs. Then, we compare the similarities between the real and randomly generated graphs to determine how well each model (and its parameters) fits for the empirically collected data. The study was conducted for 42 microservice applications, which were attributed to the power law degree distribution with the strictest conditions according to Table 1.

5.1 Structural Models Identification

A large percentage of the applications exhibited skewed degree distribution. Thus, five random graph models, which we believe describe applications that model scale-free networks, are chosen. Distance metrics are computed for every application and the results for each each metric reveal the model types that best describe the majority of the applications.

5.1.1 Considered Models. The following models were considered to identify the architectural patterns:

(1) **Erdős-Rényi random graph (ER)** in its $G(n, m)$ and $G(n, p)$ forms was used as a baseline [13]. The number of vertices n and the number of edges m are equal to that of the application graph, whereas the probability of an edge to be included in the generated graph p varies throughout the tests.

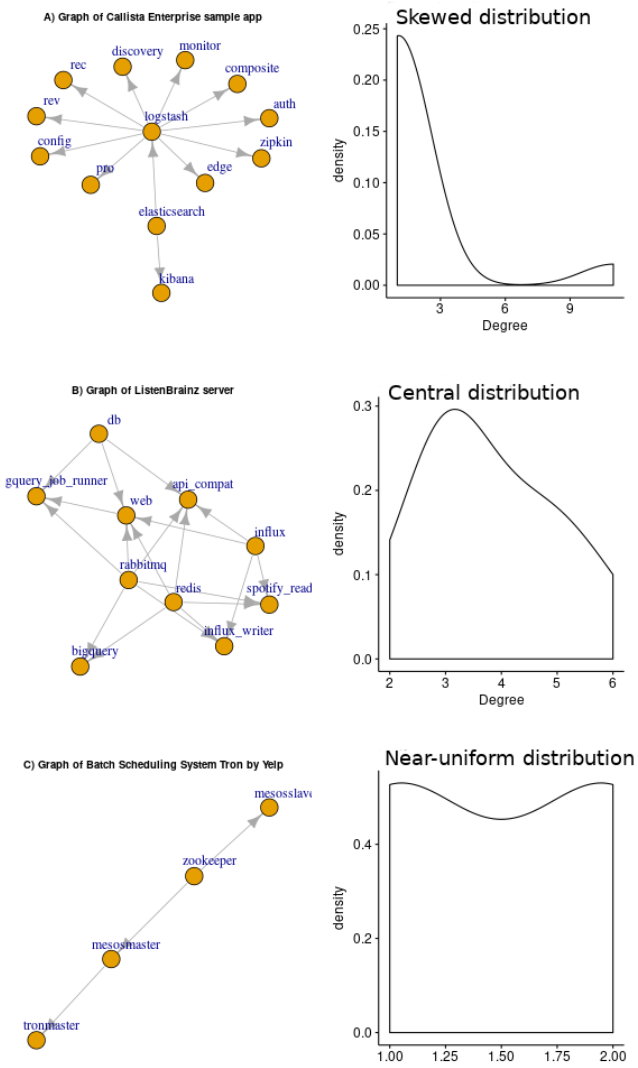

Figure 4: Samples following the proposed distributions.

(2) **Barabási-Albert (BA)** with the varying parameters: power of the preferential attachment, number of edges to add per timestep, attractiveness of vertices without edges [4];
(3) **Forest Fire (FF)** with the varying parameters: forward burning probability, backward burning ratio, number of ambassador vertices [21];
(4) **Fitness Score (FS)** that generates a graph with edge probabilities proportional to node fitness scores with the power used to generate the vector containing the fitness of each vertex as the only varying parameter [15];
(5) **Simple Power Law (SPL)** that generates a graph with a desired power law degree distribution varying only the in-degree and outdegree power law exponents [10, 15].

We used the the R package **igraph** [12] to implement these models.

5.1.2 Approach. The identification of a structural model for a single application graph starts with the generation of multiple random graphs (300) for each discussed model type (five types) with all

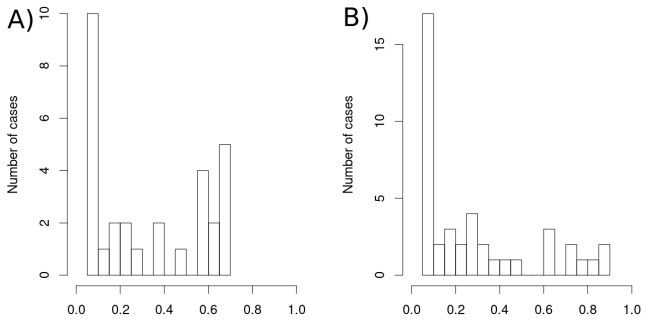

Figure 5: Distribution of the cases with the optimal parameters' values over the values of *Power of preferential attachment* (BA)

possible combinations of model parameter values *from the meaningful subspace* determined by the preliminary experiments. Such parameters as the number of vertices/edges are taken directly from the application graph.

All distance metrics are computed, for each pair of an application graph with one of the generated random graphs. Each distance metric (e.g., Hamming) for the given model type (e.g., ER) and the current set of model parameters (e.g., $n = 17$ and $p = 0.5$) is computed as the average of all pairwise distance values between the application graph and the random graphs generated from that model. Averaging ensures the stability of the results.

We tuned the model parameters via running our approach with different parameter limits multiple times. We adjusted the limits of each parameter by studying the form of the distribution of the cases with the minimal value of a distance metric over each parameter's values. If the histogram is skewed, it might be necessary to increase the upper boundary on the parameter and continue the tuning.

An experiment with 30 application graphs showed that the upper bound on the BA model's parameter *Power of preferential attachment* originally set to values from the interval $[0.05, 0.7]$ was too small as the number of cases with the optimal parameters' values increased to the end of the interval (see plot **A** in Figure 5). With the upper bound of the same parameter increased to 0.9 for the experiment involving the full set of 42 application graphs, we did not observe any increase in the number of cases towards the end of the interval (see plot **B** in Figure 5). Hence, with an exhaustive search being unfeasible, the parameters' bounds tuned with this method, cover random graphs models close to real application graphs.

5.1.3 Test Settings. The pilot test covered 30 applications. Then, we conducted three tests on 42 applications: The first returned results for all 42 applications, the second returned results for 41 applications, and the last one only for 36 applications. The last test was conducted for the case of undirected graphs; the partial results returned are due to particular distance metrics relying in their computation on matrix invertibility, which is not always the case for the given data set. Further, we discuss test settings and results of the second experiment as it covers all 42 applications. The bounds on the parameters' values are given in Table 2. The number of random graphs generated for each model type and each parameters values combination is 300. The number of vertices for each experiment was taken directly from the application graph.

Model type	Parameter	Start[a]	End[a]	Step[a]
ER $G(n, m)$	Edges number	-	-	-
ER $G(n, p)$	Edge inclusion prob.	0.05	0.65	0.05
BA	Power of the preferential attachment	0.05	0.90	0.05
	Number of edges to add per timestep	1	2	1
	Attractiveness of vertices with no edges	0.01	3.5	0.01
FF	Forward burning probability	0.05	0.65	0.05
	Backward burning ratio	1	3	1
	Number of ambassador vertices	1	2	1
FS	Power to generate fitness vector	2	3.5	0.1
SPL	Power law expon. of the out-degree distr.	2	3	0.1
	Power law expon. of the in-degree distr.	2	3	0.1

[a] "-" value is taken from the application graph.

Table 2: Studied parameter values

Distance type	Random Graphs Model Types					
	ER $G(n, m)$	ER $G(n, p)$	BA	FF	FS	SPL
Degree Centrality	0 0.00%	1 2.38%	**29 69.05%**	3 7.14%	0 0.00%	*9 21.43%*
Closeness Centrality	0 0.00%	3 7.14%	**32 76.19%**	3 7.14%	1 2.38%	3 7.14%
Between. Centrality	0 0.00%	**26 61.90%**	*11 26.19%*	0 0.00%	4 9.52%	1 2.38%
Edge Difference	0 0.00%	**32 76.19%**	*9 21.43%*	0 0.00%	1 2.38%	0 0.00%
Graph Diffusion	0 0.00%	1 2.38%	**38 90.48%**	0 0.00%	2 4.76%	1 2.38%
Hamming	0 0.00%	**32 76.19%**	*10 23.81%*	0 0.00%	0 0.00%	0 0.00%

Table 3: Cases with minimal network distance

5.1.4 Results. Network distance metrics were used to determine which one of the studied model types allows us to generate random graphs that are close to the real applications. Each metric captures different structural properties, e.g., *Degree Centrality-based distance metric* tends to mark graphs having close degree distributions as similar, whereas *Edge Difference distance metric* is small for pairs of graphs that have similar connections. These differences between metrics become apparent when looking at Table 3. Here, each row corresponds to one of the network distance types, and each column contains the number and percentage of cases with the minimal distance to the random graphs generated with the model type specified in the column header.

Since the distances were averaged over 300 generated graphs for each selected application graph from the dataset, the analysis of the cases with larger network distances is not provided as the observed gap between the model exhibiting the minimal distance and the model with the second smallest distance was higher than what would be meaningful to consider.

BA excels at capturing structural characteristics used for comparison by *Degree Centrality*, *Closeness Centrality*, and by *Graph Diffusion distance*. ER in its $G(n, p)$ form shows good results for *Betweenness Centrality*, *Edge Difference distance*, and *Hamming distance*. However, BA is still in second place with 11, 9, and 10 cases out of 42 for these distance types correspondingly. In contrast, ER in its $G(n, p)$ form has less than 6 cases in total marked as similar to real graphs by *Degree Centrality*, *Closeness Centrality*, and *Graph Diffusion distance*. Hence, BA captures the properties of the microservice applications structure nicely.

Recalling that the 42 application graphs selected for this study exhibited power law-like degree distribution, we might find it significant that for some metrics, numerous cases result in the ER in its $G(n, p)$ form. Essentially that means that a combination of BA with ER in its $G(n, p)$ form could capture the structural properties of microservice applications better than each of these model types individually. Such combinations can be enabled by generative models of graphs acquired with machine learning techniques [8, 22].

Nevertheless, further application-wise study of minimal network distances demonstrates that *Edge Difference distance* values for different models vary weakly; in 32 cases this type of distance demonstrated the smallest variability when computed for different models. Thus, we select the BA type as the best representative type for microservice application graphs.

5.2 Structural Model Generation

We then proceeded to create models that best fit the structures of our dataset. Studying the parameters of the BA model leading to minimizing the network distances shows that the change only in two parameters influences how close the generated graph is to the real one. These parameters are *power of preferential attachment*, α, and *attractiveness of vertices with no edges*, a. According to BA, a single vertex is added to the graph at each time step; a new vertex is attached to old vertices with one or more edges. The probability of i^{th} vertex to be chosen is given by $P_i = d_i^{\alpha} + a$, where d_i is the in-degree of this vertex. As we see, higher values of α favor vertices with more connections, whereas higher a values give vertices with no connections a chance to establish new ones.

Study of parameters α and a distributions for graphs with minimal network distances from the Subsection 5.1.4 allowed us to find two perspective intervals for each of these: $\alpha \in [0.01; 0.10] \cup [0.80; 1.00]$, $a \in [0.00; 0.05] \cup [3.00; 3.50]$. For each interval marked either as *LOW* or *HIGH*, a value close to its middle was selected, then four possible combinations of these values were acquired to generate example random graphs according to BA. Parameter *edges to add per time step* was set to 1. Generated samples with 18 vertices are shown in Figure 6.

Visual study shows that sample **B** in Figure 6 corresponds to the applications that rely on the common logging service, whereas sample **C** represents an application with several auxiliary services used, e.g., to maintain configurations. Sample **D** in that sense is close to applications organized in the conventional multi-tier fashion. Sample **A** in Figure 6 also finds peers among microservice applications—these exhibit highly-centralized hierarchical architectures with most of the services using the configuration service.

6 DISCUSSION

The above results lead to several observations on the structure of microservice applications and how it could be used to assure software quality attributes.

6.1 Implications of the Microservice Applications Structure

Studying 103 open-sourced Docker Compose configuration files discovered the prevalence of microservice applications with *a power law distribution of degrees* in the application graph. This structural feature implies the presence of one or several highly-connected microservices. Such a microservice application design pattern might lead to highly vulnerable applications in case microservices with a high number of connections implement a critical functionality.

In some cases, the microservice with the highest number of connections is just a logging service, hence its failure won't influence SLOs. Thus, structural analysis and modeling of microservice applications *should be enhanced with the analysis of the functional context* such that critical microservices are clearly recognized and are not mixed with ones that are not critical but are still highly relied upon. Such information can be used to ensure that the availability, throughput and resource requirements are satisfied by helping decide the appropriate number of critical microservices' replicas.

Among several graph generation models studied, the BA-model demonstrated an ability to capture the degree distribution of the microservice application using relatively small intervals of values for its parameters *power of preferential attachment* and *attractiveness of vertices with no edges*. Changing these parameters means modifying the number of connections that few nodes have (first parameter) and changing the number of nodes central to some local clusters of nodes (second parameter). A high value of the parameter *attractiveness of vertices with no edges* allows us to model fairly complex graphs with several "centers of attraction".

The study of network distances between generated random graphs and 42 microservice applications graphs underlines that one model cannot convey all the properties of the microservice application structure. This can be solved via analytical models that generate random graphs exhibiting characteristics of several models: consider similar work by Solé et al. [30] or by learning a deep generative graph model on a representative set of examples [22].

Both simple and hybrid random graph models can be employed to synthesize structures that correspond to real microservice applications. Varying the parameters of such models would enable capturing the peculiarities of a microservice application's structure. As one can select the number of vertices and edges for such models arbitrarily, the absence of large open-sourced microservice applications does not hinder the design and evaluation of algorithms utilizing in some way the information on the applications' structures. However, with the simplifications that could be made when identifying the appropriate random graph model (e.g., omitting information on types of services), it may become necessary to validate the model manually by developing a sample large-scale microservice application with limited functionality.

The analysis of the microservice application's structures in the paper is based on the degree of graph vertices. This could be viewed as a limiting factor as the graph abstraction offers a rich set of parameters to study the microservice structure in-depth, e.g., vertex connectivity or isoperimetric number. For example, one could think of studying the vertex connectivity of the microservice applications' graphs to identify the cornerstone services whose removal, say due to failure, damages the functionality of the application. An isoperimetric number can be used in studies of potential bottleneck services. Consideration of these parameters was deemed beyond the scope of this paper.

The conducted structural analysis makes a strong assumption that the application is static, which in practice does not always hold true. Addition and removal of microservices over time is a

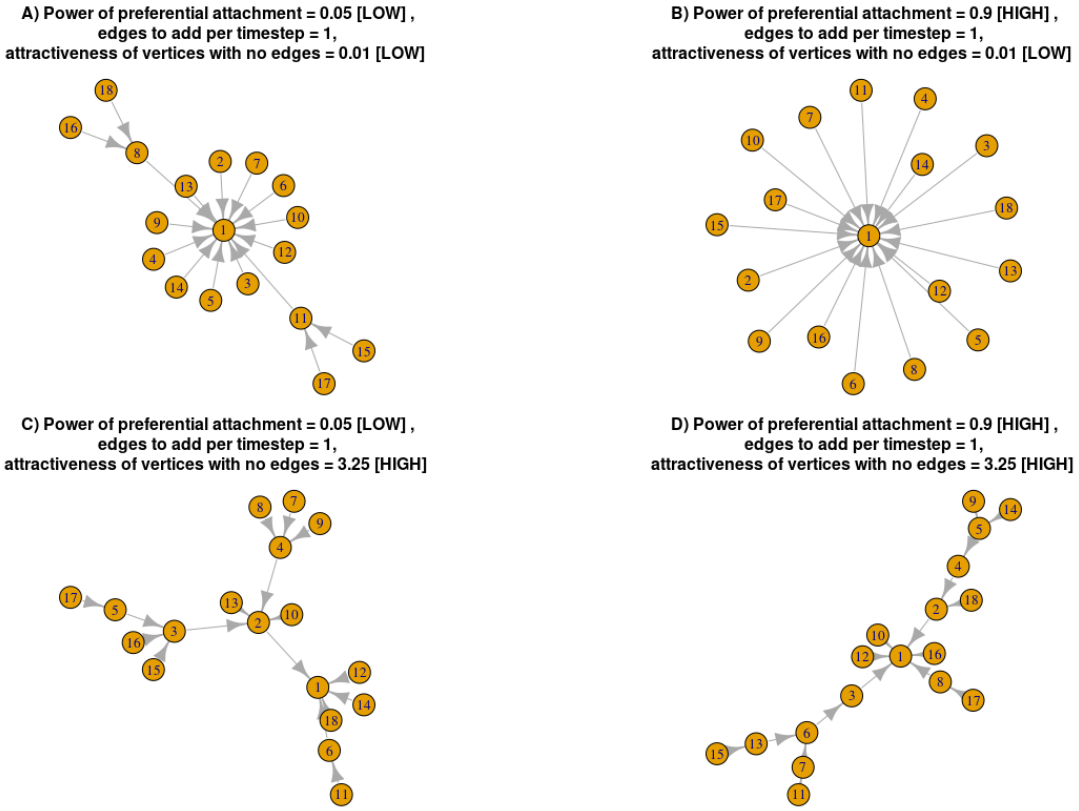

Figure 6: Random Graphs generated using BA with four parameter. Number of nodes: 18.

normal practice for such applications. Dynamic graph analysis of the microservice application will lead to models that capture the evolution of the application. In turn, such models could contribute to increasing the accuracy of predictive autoscaling by incorporating the knowledge of potential structural changes in the model.

6.2 Application Structure towards assuring Software Quality Attributes

Knowing an application's structure can contribute to quality assurance of the application across the software life cycle:

Scalability and Availability. Revealing the relationship between scaling events and applications' capacity will lead to the fine tuning of the scaling actions; instead of individual scaling actions one might speak of *scaling action cascades* directed by the structure of an application and capacities of microservices. We believe that the adaptation of the microservice applications to changing workloads can be improved by including the application structure into the set of autoscaling parameters. Such improvements for real elastic microservice applications hosted in the cloud can result in better quality of service and budget savings, therefore it seems necessary to consider the application structure when scaling.

Testability and Correctness. We identified and replicated the applications' architectures. To this end, realistic benchmarks can be created using these models as a generic template. The templates can be used in the testing process for the product or for cases that the product acts as an input for other applications. Additionally, computationally expensive quality assurance methodologies, such as formal verification, could be better targeted towards the various soft points in a topology of an application.

Security and Reliability. The identification of the *weakest link* service with the most services that depend on it can help to make precautions for protecting the applications in advance or making changes in the infrastructure to make it safer from attacks. More specifically, certain rollback policies can be applied based on the service dependencies in case they go offline.

Performance efficiency. The *weakest link* services can be deployed based on their connectivity. Certain resources can favour certain types of services to achieve better response times and thus better performance. The configuration of a service-container can be set to allow for more hardware resources on critical services than on less critical ones.

Adaptivity. From a self-adaptive systems perspective, being able to create and analyze models of one's composition is a crucial self-* property that can be used to analyze and plan adaptation such that various quality attributes (or setpoints/goals) are satisfied.

Finally, the application topology reveals the *strongest link*. By making certain design choices that will shift the load from the strongest to the weakest service can help towards the application quality, as well.

7 CONCLUSION AND FUTURE WORK

The study discovered degree distributions that are widely-present in graphs of 103 open-sourced microservice applications: power law, uniform, and normal. Looking closer at 42 applications that exhibited power law-like degree distribution allowed us to discover that BA-based random graphs capture the structure of real microservice applications well. By employing this model, one can synthesize random graphs with a large number of vertices that capture the structural properties of microservice applications.

The study paves the way towards larger and systematic empirical studies of how microservice applications tend to be structured, resulting in new heuristic algorithms for improved scaling, self-protection from targeted attacks, testing and system administration. Revealing and generating models based on their connectivity, while viewing an application as a directed graph of services, can be very helpful for application evolution.

The following future research directions appear to have significant utility in microservice applications deployment and management: customized analytic and machine learning-based graph models to generate random graphs; extension of the structural modeling and analysis with microservice types; extending graph models capturing properties of microservice applications with other graph characteristics and building dynamic graph models to predict structural changes. The main limiting factor for the research of microservice application structures is the novelty of the concept and limited public availability of real microservice applications. With the continuing adoption of the microservice architecture for cloud-native applications, more data would become available in public repositories and more mining-based research can be done. With more publicly available knowledge, we aim to explore further types of applications that use certain programming languages and frameworks to reveal even more aspects of the status quo of software products.

ACKNOWLEDGMENTS

This work was supported by AWS research program Cloud Credits, STRATUS, a project funded by New Zealand's Ministry of Business, Innovation and Employment (MBIE), the Natural Sciences and Engineering Research Council of Canada (NSERC) and Canada's New Brunswick Innovation Fund (NBIF). We also thank Stephen MacKay for his careful proofreading and editing the paper to improve its quality. We also thank anonymous reviewers for their comments which we tried to address in the final version of the paper.

REFERENCES

[1] 2020. Docker compose files to analyze structural patterns of containerized microservice applications. https://doi.org/10.5281/zenodo.3573846. [Online; accessed 8-June-2020].
[2] 2020. The Official YAML Web Site. https://yaml.org/. [Online; accessed 26-March-2020].
[3] Réka Albert and Albert-László Barabási. 2002. Statistical mechanics of complex networks. *Rev. Mod. Phys.* 74 (Jan 2002), 47–97. Issue 1.
[4] Albert-László Barabási and Réka Albert. 1999. Emergence of Scaling in Random Networks. *Science* 286, 5439 (1999), 509–512.
[5] Albert-László Barabási and Márton Pósfai. 2016. *Network science.* Cambridge University Press, Cambridge. http://barabasi.com/networksciencebook/
[6] R. Birke, J. F. Perez, Z. Qiu, M. Borkqvist, and L. Y. Chen. [n.d.]. sPARE: Partial Replication for Multi-tier Applications in the Cloud. *IEEE Transactions on Services Computing* ([n. d.]), 1.
[7] J. Bogner, J. Fritzsch, S. Wagner, and A. Zimmermann. 2019. Microservices in Industry: Insights into Technologies, Characteristics, and Software Quality. In *2019 IEEE International Conference on Software Architecture Companion (ICSA-C).* 187–195.
[8] Aleksandar Bojchevski, Oleksandr Shchur, Daniel Zügner, and Stephan Günnemann. 2018. NetGAN: Generating Graphs via Random Walks. In *ICML.*
[9] Phillip Bonacich. 1987. Power and Centrality: A Family of Measures. *Amer. J. Sociology* 92, 5 (1987), 1170–1182.
[10] Fan Chung and Linyuan Lu. 2002. Connected Components in Random Graphs with Given Expected Degree Sequences. *Annals of Combinatorics* 6, 2 (01 Nov 2002), 125–145.
[11] Rhys Compton, Eibe Frank, Panos Patros, and Abigail Koay. 2020. Embedding Java classes with code2vec: improvements from variable obfuscation. In *IEEE/ACM 17th International Conference on Mining Software Repositories (MSR 2020).* ACM.
[12] Gabor Csardi and Tamas Nepusz. 2006. The igraph software package for complex network research. *InterJournal* Complex Systems (2006), 1695. http://igraph.org
[13] Paul Erdős and Alfréd Rényi. 1959. On Random Graphs I. *Publicationes Mathematicae (Debrecen)* 6 (1959 1959), 290–297.
[14] M. Fazio, A. Celesti, R. Ranjan, C. Liu, L. Chen, and M. Villari. 2016. Open Issues in Scheduling Microservices in the Cloud. *IEEE Cloud Computing* 3, 5 (Sep. 2016), 81–88.
[15] K.-I. Goh, B. Kahng, and D. Kim. 2001. Universal Behavior of Load Distribution in Scale-Free Networks. *Phys. Rev. Lett.* 87 (Dec 2001), 278701. Issue 27.
[16] W. Hasselbring and G. Steinacker. 2017. Microservice Architectures for Scalability, Agility and Reliability in E-Commerce. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW).* 243–246.
[17] Adery C. A. Hope. 1968. A Simplified Monte Carlo Significance Test Procedure. *Journal of the Royal Statistical Society. Series B (Methodological)* 30, 3 (1968), 582–598. http://www.jstor.org/stable/2984263
[18] Docker Inc. 2020. Compose file version 3 reference. https://docs.docker.com/compose/compose-file/. [Online; accessed 26-June-2020].
[19] Steffen Kächele and Franz J. Hauck. 2013. Component-based Scalability for Cloud Applications. In *Proceedings of the 3rd International Workshop on Cloud Data and Platforms (CloudDP '13).* ACM, New York, NY, USA, 19–24.
[20] Robert V. Krejcie and Daryle W. Morgan. 1970. Determining Sample Size for Research Activities. *Educational and Psychological Measurement* 30, 3 (1970), 607–610.
[21] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. 2005. Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining (KDD '05).* ACM, New York, NY, USA, 177–187.
[22] Yujia Li, Oriol Vinyals, Chris Dyer, Razvan Pascanu, and Peter Battaglia. 2018. Learning Deep Generative Models of Graphs. arXiv:cs.LG/1803.03324
[23] X. Liu, J. Heo, and L. Sha. 2005. Modeling 3-tiered Web applications. In *13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems.* 307–310.
[24] S. Malkowski, M. Hedwig, and C. Pu. 2009. Experimental evaluation of N-tier systems: Observation and analysis of multi-bottlenecks. In *2009 IEEE International Symposium on Workload Characterization (IISWC).* 118–127.
[25] G. Márquez, M. M. Villegas, and H. Astudillo. 2018. An Empirical Study of Scalability Frameworks in Open Source Microservices-based Systems. In *2018 37th International Conference of the Chilean Computer Science Society (SCCC).* 1–8.
[26] V. Podolskiy, M. Mayo, A. Koay, M. Gerndt, and P. Patros. 2019. Maintaining SLOs of Cloud-Native Applications Via Self-Adaptive Resource Sharing. In *2019 IEEE 13th International Conference on Self-Adaptive and Self-Organizing Systems (SASO).* 72–81.
[27] Nigel Poulton and Pushkar Joglekar. 2019. *The Kubernetes Book* (fourth ed.).
[28] J. F. Pérez, L. Y. Chen, M. Villari, and R. Ranjan. 2018. Holistic Workload Scaling: A New Approach to Compute Acceleration in the Cloud. *IEEE Cloud Computing* 5, 1 (Jan 2018), 20–30.
[29] S. S. Shapiro and M. B. Wilk. 1965. An analysis of variance test for normality (complete samples)†. *Biometrika* 52, 3-4 (1965), 591–611.
[30] Ricard V Solé, Romualdo Pastor-Satorras, Eric Smith, and Thomas B Kepler. 2002. A model of large-scale proteome evolution. *Advances in Complex Systems* 05, 01 (2002), 43–54.
[31] M. A. Stephens. 1974. EDF Statistics for Goodness of Fit and Some Comparisons. *J. Amer. Statist. Assoc.* 69, 347 (1974), 730–737.
[32] Q. Wang, Y. Kanemasa, J. Li, D. Jayasinghe, T. Shimizu, M. Matsubara, M. Kawaba, and C. Pu. 2013. Detecting Transient Bottlenecks in n-Tier Applications through Fine-Grained Analysis. In *2013 IEEE 33rd International Conference on Distributed Computing Systems.* 31–40.
[33] Qingyang Wang, Y. Kanemasa, J. Li, D. Jayasinghe, T. Shimizu, M. Matsubara, M. Kawaba, and C. Pu. 2013. An Experimental Study of Rapidly Alternating Bottlenecks in n-Tier Applications. In *2013 IEEE 6th International Conference on Cloud Computing (CLOUD),* Vol. 00. 171–178.