Institute of Architecture of Application Systems

Serverless Parachutes: Preparing Chosen Functionalities for Exceptional Workloads

Vladimir Yussupov, Uwe Breitenbücher, Michael Hahn, Frank Leymann

Institute of Architecture of Application Systems,
University of Stuttgart, Germany,
{yussupov, breitenbuecher, hahn, leymann}@iaas.uni-stuttgart.de

BibTeX:

```
@inproceedings{Yussupov2019_ServerlessParachutes,
  author    = {Vladimir Yussupov and Uwe Breitenb{\"u}cher and Michael Hahn
               and Frank Leymann},
  title     = {{Serverless Parachutes: Preparing Chosen Functionalities for
               Exceptional Workloads}},
  booktitle = {Proceedings of the 2019 IEEE 23rd International Enterprise
               Distributed Object Computing Conference (EDOC 2019)},
  publisher = {IEEE},
  year      = 2019,
  month     = oct,
  pages     = {226--235},
  doi       = {10.1109/EDOC.2019.00035}
}
```


Universität Stuttgart
Germany

Serverless Parachutes: Preparing Chosen Functionalities for Exceptional Workloads

Vladimir Yussupov, Uwe Breitenbücher, Michael Hahn, and Frank Leymann
Institute of Architecture of Application Systems
University of Stuttgart, Universitätsstraße 38, Stuttgart, Germany
Email: [vladimir.yussupov, uwe.breitenbuecher, michael.hahn, frank.leymann]@iaas.uni-stuttgart.de

***Abstract*—Function-as-a-Service (FaaS) is an emerging cloud service model that enables composing applications using arbitrary, small, and event-driven code snippets managed by cloud providers and that can be scaled to zero. The scalability properties of FaaS look attractive for handling rare or unexpected high loads that affect only particular functionalities of the application. However, deciding on the component granularity upfront or reengineering the architecture of an entire application for rare workloads is often a very difficult challenge or even infeasible. In this work, we introduce a method that prepares annotated functionalities for handling rare workloads by automatically extracting them from the source code of the application and additionally deploying them as FaaS functions, while keeping the original application's functionalities and architecture unchanged. In this way, the benefits of FaaS can be leveraged without the need to reengineer the application only for rare cases. We validate our method by means of a prototype, evaluate its feasibility in a set of experiments, and discuss limitations and future work.**

***Keywords*-Serverless, FaaS, Function-as-a-Service, Scalability, Failover, Annotation**

I. Introduction

Cloud computing is an essential building block in the IT landscape of many modern enterprises [1]. The widespread adoption of cloud computing is the result of its substantial benefits, such as elasticity and continuous deployment at global scale [2]. One typical example is the usage of cloud compute resources instead of buying the hardware. Unlike traditional infrastructure investments, it is notably easier to deal with redundant compute resources in the cloud, which helps reducing costs. Likewise, using higher abstraction layers, e.g., platform or software, helps further reducing the maintenance burden and optimizing development and operations processes. In the context of cloud, the *serverless* computing paradigm further abstracts away the infrastructure by focusing on developing applications using only provider-managed components. While being a broader concept, the term serverless is often used to describe a particular cloud service model, called Function-as-a-Service (FaaS). FaaS hides the infrastructure by allowing to deploy arbitrary, fine-grained, event-driven code snippets [3], [4], which are fully managed by providers. As a result, the scaling is automatically performed by providers, and, moreover, FaaS deployments can be *scaled to zero*, allowing to pay only for the actual usage of functions, without considering the idle time [5]. The prominent examples of public FaaS offerings are AWS Lambda [6] and Microsoft Azure Functions [7].

Event-driven, FaaS-hosted functions are attractive for dealing with rare or unpredictable high workloads, since functions are automatically scaled by providers without limits. However, (i) deploying every functionality to FaaS is not often appropriate and can result in complex, unmanageable application architectures with thousands of components. For example, deploying two functionalities interacting via local calls to FaaS results in a communication overhead, as both now need to interact in an event-driven manner. Additionally, (ii) choosing the right components granularity, e. g., whether it is a FaaS-hosted function or the part of a PaaS-hosted component, is a highly non-trivial challenge [8]–[10]. Moreover, (iii) often it is even not possible to use FaaS for certain functionalities as it strongly impacts the application's architecture. Therefore, keeping crucial application functionalities available via FaaS requires big efforts and comes with far-reaching design decisions—ironically, only to prepare the application for hypothetical workloads that might never occur.

In this paper, we tackle this issues and present a method for preparing crucial application functionalities for exceptional workloads by leveraging the advantages of the FaaS cloud service model without the need to change the actual architecture of the application. In our method, parts of the application's source code are first enriched with annotations, which enables the automated extraction and generation of FaaS-deployable function bundles called *serverless parachutes*. These bundles are deployed to FaaS offerings as backup routes for handling exceptional workloads, whereas original functionalities of the application and the application itself are used as-is in regular cases. The most important benefit of the method is that it does not require reengineering the original application's architecture or deciding on the right granularity of scalable components upfront. To validate our method, we implement a prototype supporting the automatic extraction, generation, and deployment of serverless parachutes. Furthermore, we perform an evaluation by conducting a set of experiments, discuss the results, and outline future research directions.

The remainder of this paper is structured as follows. In Section II, we define the problem statement, introduce a running example, and discuss the related work. We introduce the serverless parachutes method in Section III and elaborate on the prototype and results of the evaluation in Section IV and Section V. Finally, Section VI concludes this paper.

II. Problem Statement and Related Work

In this section, we describe the relevant background, define the problem statement and introduce a running example. Furthermore, we elaborate on the related work and its applicability to the highlighted problems.

A. Serverless Computing and Function-as-a-Service

Historically, the term *serverless* has been used in different contexts, from peer-to-peer communication and client side-only software to RFID protocols [11]–[13]. In the context of cloud, the concept of *serverless computing* describes a programming model and architecture that focus on building applications by composing functionalities that are executed in the cloud without the need to manage the underlying infrastructure [3]. For example, (Mobile) Backend-as-a-Service (BaaS) and Function-as-a-Service (FaaS) service models fit into this category. While the former focuses on combining third-party APIs to avoid having a traditional backend system, FaaS focuses on using provider-managed, arbitrary, event-driven, and typically, stateless, code snippets as application building blocks. Both BaaS and FaaS abstract away the infrastructure and reduce the management effort and complexity, which gives the impression that servers do not exist. However, the reduced management efforts in no way mean that servers are absent. Instead, tasks such as resource provisioning, monitoring, scaling, and fault tolerance become a burden of cloud service providers [3], [4]. Another important property of FaaS is that the application logic can be scaled to zero resulting in a new cost model where the application owner is not charged for idle periods, which is not the case, e.g., in Platform-as-a-Service deployments. Despite the advantages, the FaaS service model is typically not enough on its own for composing serverless applications and has certain limitations [14], [15]. For instance, stateless and ephemeral FaaS functions have limited lifetime and often need store the state in shared storage systems such as AWS S3. These limitations are among the factors influencing the decision on choosing the optimal component granularity and the cloud service model.

B. Problem Statement

Failures are imminent for software at any scale [16]–[18]. Problems such as hardware failures or workload changes lead to resource exhaustion [18] and might cause a sudden system outage. Ironically, fault tolerance and recovery strategies might also lead to a system failure [17], e. g., Amazon's outage, which happened in 2017 due to increased time of AWS S3 services recovery, affected recovery strategies of other AWS services. Often, crucial functionalities must remain available in exceptional cases, which lead to sudden, unrecoverable resource exhaustion, e. g., disasters or improper infrastructure planning. For instance, in cases of disasters, functionalities of disaster management systems supporting assistance operations during or after disasters must be available [19]. Dealing with unexpected resource exhaustion is also important for hybrid clouds, e. g., the cloud bursting [20] approach allows bursting resources into public clouds to handle high load spikes.

Interestingly, several research works [19], [21] describe the auto-scaling and scale-to-zero features of FaaS as a suitable solution for withstanding exceptional workloads, due to the fact that availability and reliability are guaranteed by providers. If important functionalities could be made available as FaaS-hosted functions, the unexpected loads can be handled by executing these functionalities at FaaS provider's side. However, despite the scaling and availability advantages of FaaS, several important factors can complicate the process of moving chosen functionalities to FaaS or even prevent it completely. Firstly, adoption of FaaS requires developers to employ a specific programming model where chosen functionalities interact in an event-driven fashion and comply with the requirements of FaaS providers, e. g., limited execution time. Thus, parts of the original application's architecture related to chosen functionalities need to be changed to comply with the new requirements, which introduces implementation overhead only for handling rare, exceptional workloads. Additionally, moving certain functionalities to FaaS might not be feasible cost- and effort-wise, e. g., moving tightly-coupled and stateful components introduces implementation and performance overhead.

Another important factor that increases the development complexity is the need to decide on application components granularity upfront, which is a non-trivial challenge [8], [22]. For instance, several studies investigated challenges for migrating applications to microservice architectures [23] in industry [24], [25] and the described big issues include decomposing existing systems, identifying proper context boundaries, and finding the right component granularity. Moreover, using finer-grained components for every functionality increases the architecture's complexity and introduces the communication overhead, e. g., microservice architectures typically have communication overhead compared to monoliths [26]. In complex systems consisting of multiple fine-grained components, the communication overhead is even higher due to a larger number of endpoints, which results in more intermediary hops needed. Likewise, applying the FaaS paradigm to every possible functionality or deciding on the right granularity upfront can be problematic and might cause additional problems instead of solving existing ones. Therefore, the decision which functionalities can be moved to finer-grained service models needs to be taken carefully as it may harm the system in the future.

Since choosing the right granularity for components upfront and reengineering an entire application only to prepare a subset of crucial functionalities for exceptional workloads requires considerable efforts from development and operations, it is often not feasible to tackle rare loads or failures. Therefore, the research question underlying this work is *"How to support developers in preparing crucial application functionalities for handling exceptional workloads by utilizing the FaaS cloud service model, without the need to change the application's architecture?"*. Our goal is to keep the original application as-is for regular cases, while using FaaS only in exceptional cases. To keep the entry barrier for developers as low as possible and to minimize the development overhead, the proposed method has to provide automation support where it is possible.

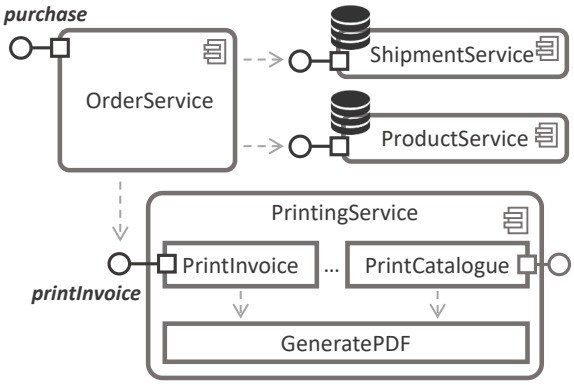

Fig. 1. Excerpt of a system architecture with four components

C. Motivating Scenario

A simplified UML component diagram in Figure 1 demonstrates an excerpt of a system architecture consisting of four loosely-coupled components, namely *OrderService*, *ShipmentService*, *ProductService*, and *PrintingService* that communicate, e. g., via REST API calls. For the sake of brevity, we omit the details about the internal structure of all components except the PrintingService component, which groups together all printing-related functionalities such as *PrintInvoice*, *PrintCatalogue*, and the internally-used *GeneratePDF* functionality. The PrintingService component is not stateful and does not require a resource layer for processing various printing-specific requests via respective exposed interfaces. Furthermore, since the PrintingService component comprises interdependent functionalities related only to printing domain, it is not further split into finer-grained, loosely-coupled components. For instance, to produce respective PDF documents, both PrintInvoice and PrintCatalogue functionalities depend on the internal GeneratePDF functionality, which provides means to generate PDF documents based on the given input.

In practice, this generic example can reflect a part of an application dealing with, e. g., processing orders in a web store which has all components implemented, e. g., as Java web applications. As an example, the purchase operation can be invoked via the interface called *purchase* provided by the OrderService component. This interface can be used only if the three remaining components are available and one of the required interfaces, namely *printInvoice*, is provided by the PrintingService component. In case of an unexpected failure of the PrintingService component, e. g., sudden spike of purchase requests on Christmas, further processing becomes not possible, leading to a full system failure. However, if the printInvoice interface, which depends on the PrintInvoice and GeneratePDF functionalities, is available in an event of failure, orders can still be processed. In the following sections, we introduce a method for automatically preparing chosen functionalities, e. g., the functionality provided via the printInvoice interface, for unexpected loads using the FaaS service model and without reengineering the original application, which allows it to function as-is in regular cases.

D. Related Work

In this subsection, we elaborate on related work and discuss its applicability to the discussed problem statement.

Khadka et al. [27] introduce serviciFi, a comprehensive method for migrating legacy code to SOA. This method comprises multiple phases, including (i) project initiation, (ii) identification of candidate services and their further specification, (iii) a construction and testing phase, and (iv) a deployment, monitoring, and management phase. The overall method is assembled from multiple fragments of existing SOA development methods and combines numerous manual and automated activities. While the phases of this method can serve as a good basis, the overall method focuses on a different problem and is too complex for applying it to our use case. Moreover, the presence of manual activities in phases, e. g., an extracted service might need to be refactored before the next phase, makes it infeasible to apply this method for certain functionalities in rare cases.

Frey and Hasselbring [28] address the problem of migrating existing software to the cloud. A model-based approach called CloudMIG begins with extracting the model of an existing application, selecting the target provider's model, i. e., a so-called Cloud Environment Model (CEM), and generating three artifacts, namely a target architecture, a mapping model, and a cloud environment's constraints violation model. Afterwards, the generated artifacts can be adapted, evaluated, and used for actual transformation of the existing application for the cloud. This approach is not suitable for our needs due to several reasons: (i) the approach focuses on migrating the entire system without the intention to keep the original application working as-is, (ii) the transformation from the generated architecture to the chosen cloud environment is not discussed, and, according to authors not planned to be implemented in the near future.

Kwon and Tilevich [10] introduce a collection of cloud refactoring techniques which help to automate the process of moving centralized applications to the cloud. Two main components underlying presented techniques are recommendation and refactoring engines. The former is responsible for providing recommendations for suitability of functionalities using dynamic and static code analysis techniques. The latter helps to refactor functionalities, including extracting, wrapping them, and adding fault-handling logic. This approach focuses on extracting the functionalities and reengineering the original application's code, which is not suitable for our use case. However, the recommendation strategies can serve as a basis for our future work about functionality's suitability.

Several works [29], [30] focus on FaaSification approaches for different programming languages. The general process involves code analysis and decomposition into functional units and code that is not suitable for FaaS. Next, the functional units are translated into FaaS units followed by compilation and assembling of their dependencies. While providing some insights about the FaaS-related code extraction, these approaches are not suitable due to the focus on migrating all application's functions and modification of the original

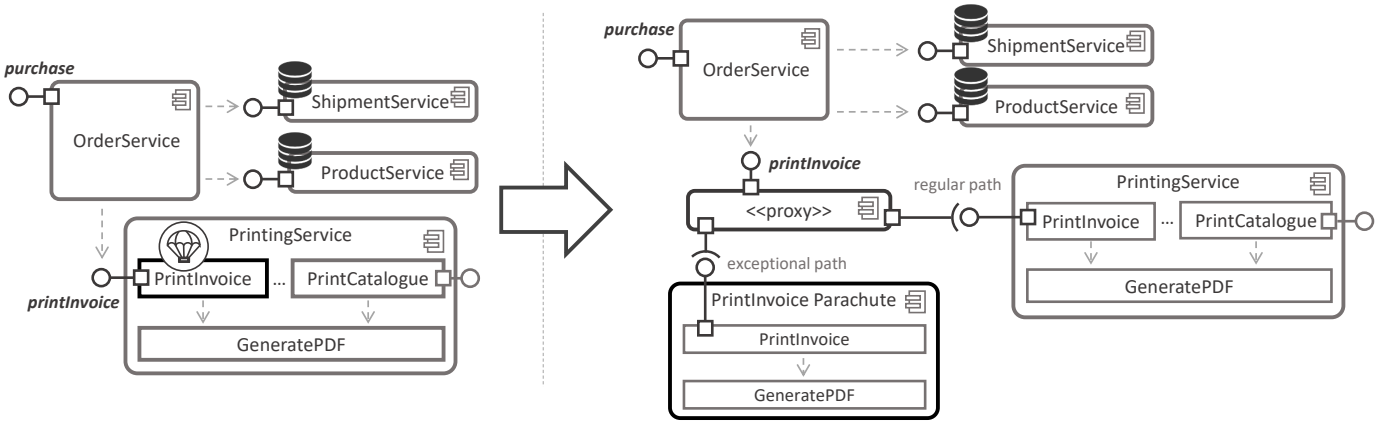

Fig. 2. The Serverless Parachutes method applied to the PrintInvoice functionality

project's structure. For Java applications, the prototypical implementation has limitations, e. g., methods located in classes that have inner classes cannot be extracted.

Perera and Perera [31] present TheArchitect, a tool for generating high-level, serverless and microservice-based architectures. The tool's input wizard collects system requirements, maps it to predefined models, generates a high-level architecture, and provides a visual representation of the result. This approach is not suitable due to several reasons: (i) the tool's output is a high-level architecture with no real implementation behind it, (ii) and analyzing and migrating chosen functionalities of an existing system is not possible.

Asghar et al. [19] discuss suitability of serverless computing for Disaster Management Information Systems (DMSI) based on their specific load characteristics. During the high activity phase, when relief operations happen after the disaster, DMSI is overloaded with a huge number of requests, whereas during the low activity phase, DMSI is almost idle. Such load characteristics make FaaS a good deployment target, since the QoS consistency will only be affected right in the beginning of the spike due to the so-called cold start [3] problem. In addition, this will allow saving time, which is required for instantiation of virtual machines or pools of containers.

The strangler pattern [24], [32] describes how to gradually migrate the original application to a chosen, modern target platform, while keeping it functioning as-is without the migrated functionalities. Eventually, the gradual removal of functionalities leads to application being migrated completely. Modifications of the original application makes implementations of this pattern not suitable for our stated problem.

The term code mobility [33] is used to describe a capability to dynamically modify bindings between the location of code and the actual code snippet. There are two forms of code mobility, namely weak and strong mobility. The former is concerned with moving the code between computational environments without moving the execution state, whereas the latter allows migrating both, the code and its execution state. In this paper, we focus on the problem of weak mobility, more specifically, on how the chosen functionality can be shipped to a new service model in an asynchronous fashion.

Hendrickson et al. [34] present OpenLambda, an open-source serverless computing platform. The authors mention that annotations identifying web resource handlers in Python's frameworks like Flask can be seen as a good starting point for decomposing legacy applications for FaaS deployments.

Numerous works [35]–[38] describe applicability scenarios for serverless computing, and FaaS in particular. The research targets various fields such as high-performance computing, artificial intelligence, or scientific workflows. While FaaS is a relatively new cloud service model, the described results of applying it to different problem domains are mainly positive.

III. The Serverless Parachutes Method

In this section, we describe the method for preparing important functionalities of existing applications to withstand rare or unexpected high loads caused by exceptional cases, without changing the architecture of the original application.

The *serverless parachutes* method's main idea is to provide developers with an intuitive and automated way of preparing crucial application functionalities for unexpected loads, without reengineering or redeveloping the original application as well as without deciding on the component granularity upfront. In the serverless parachutes method, the chosen crucial functionalities of an application are duplicated to FaaS deployments, while keeping the original application functioning as-is. A generated *proxy* component, which is decoupled from the original application and is capable of conditional routing, then redirects requests to this FaaS-hosted functionalities only in exceptional, high workload cases.

Figure 2 depicts how the method can be applied to the functionality provided via the printInvoice interface from the running example described in Section II-C. The generated PrintInvoice Parachute is a FaaS-hosted function that combines the logic of PrintInvoice and GeneratePDF sub-components to make the functionality provided via the printInvoice interface available for exceptional cases. In the meantime, regular cases are handled by the original Printing component, whose architecture remains unchanged. This approach avoids (i) reengineering the original application, i. e., the architecture of the Printing component in this example, only for exceptional

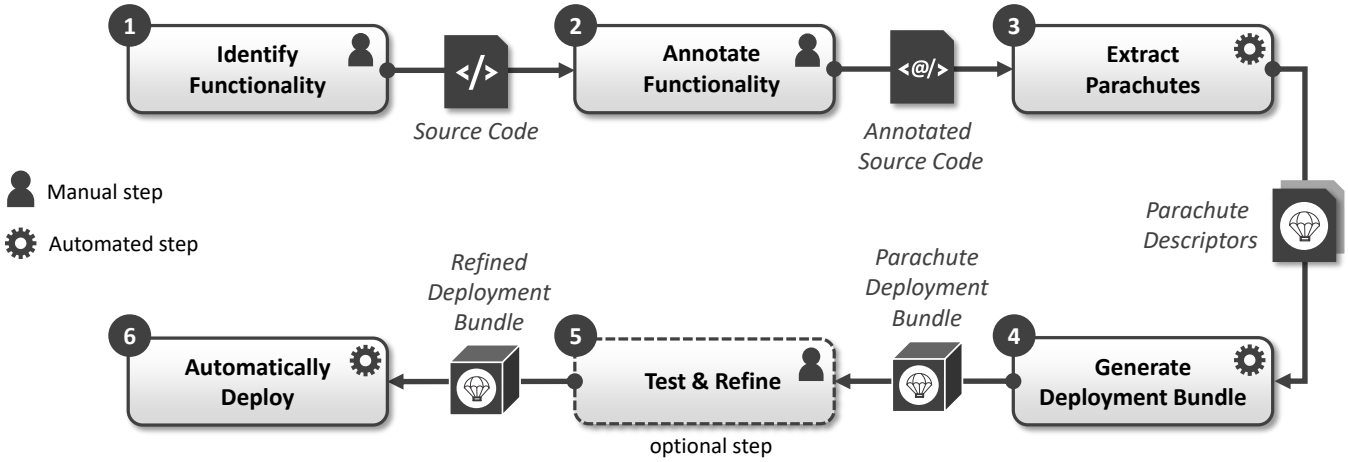

Fig. 3. A stepwise diagram of the serverless parachutes method

cases, and (ii) scaling the entire application, which leads to cost optimizations. We refer to the deployed FaaS functionality as a *serverless parachute* emphasizing on the backup nature of the duplicated deployment.

Serverless parachutes enable an efficient, fine-grained scaling by utilizing the benefits of the FaaS cloud service model for handling exceptional workloads as described in Section II. A simple example of a suitable candidate for extraction as a serverless parachute is a stateless Java servlet with no internal dependencies. To achieve such deployment duplication, both the original and the newly-created FaaS version of the functionality must conform to the original interface and be accessible by means of rerouting using a separate proxy component. The rerouting happens only if a certain condition is fulfilled, e. g., when the number of requests is above a certain threshold or when the original functionality is currently not available. The overall method of creating serverless parachutes consists of six steps, namely (i) identifying suitable functionalities, (ii) annotating them on the level of the source code, (iii) extracting the required information about the functionality, (iv) generating parachute deployment bundles for the provider of choice, (v) refining and testing resulting deployment bundles, and (vi) deploying them automatically. Figure 3 demonstrates this process including the descriptions of inputs and outputs for each step. In the following, we describe every step in more detail.

A. Step 1: Identifying Suitable Functionalities

The first step of the method, as shown in Figure 3, is to identify functionalities suitable for parachute deployments. The problem of finding suitable functionalities is not in the scope of this work. However, we briefly discuss which functionalities can be potential candidates for parachute deployments. In the next steps, we assume that the functionality is already identified and available for further processing.

Clearly, not all functionalities are suitable for migration to cloud [10]. The most suitable functionalities for a parachute deployment have several common characteristics, which correlate with the properties of the FaaS service model. More specifically, they are: (i) independent or easy to decouple, i. e., proper bounded contexts, (ii) stateless, and, (iii) preferably, finer-grained components, e. g., having small amount of distinct functionalities. A good example is a stateless Java servlet that handles requests and has no internal dependencies.

On the other hand, functionalities not suitable for parachuting, for example, (i) cannot be decoupled without introducing additional components, e. g., duplicate the used storage, or (ii) are too expensive to decouple due to critical dependencies on the source application. One example of less-suitable functionality is related to code snippets, which, when decoupled, still depend on the main application, e. g., the try-catch blocks for fault handling. Extracting such code snippets and deploying them to FaaS for handling main application's faults would require integration with the main application, which makes conditional FaaS routing in exceptional cases not feasible: the source application must be available and, moreover, will become a scaling bottleneck. In general, such tightly-coupled examples require additional reengineering and maintaining efforts, which neglects the advantages of generating parachutes for such functionalities.

B. Step 2: Annotate Functionalities

The second step of the method, as depicted in Figure 3, is responsible for enriching identified functionalities with additional meta-data by means of source code annotations. Various additional information can be specified for defining the parachutes behavior, e. g., desired routing conditions. Although the conceptual model of annotations is uniform, the actual implementations are language-specific. Depending on the programming language, annotations can be implemented differently, e. g., using Java annotations, C# attributes, or standardized commentary-based annotations. An example of such parachute annotations in Java is shown in Listing 1. The *ParachuteMethod* annotation marks the serverless parachute functionality. Routing condition properties describe at which point requests have to be rerouted to the parachute, e. g., when the main path is not available, at a specific date or certain availability threshold. Of course, this is extensible as various

Listing 1. Example usage of parachute annotations in Java

```
@ParachuteMethod(
failover = true,
overUtilizationFactor = "80%",
routeOnDate = "20191224", ...)
public Response handler(Request data) {...}
```

routing conditions are possible and not limited to the discussed ones. Moreover, depending on the desired level of control during the next steps, annotations might contain more specific directives, e. g., mapping rules for input and output parameters in typed programming languages or a custom endpoint name for a parachute deployment.

C. Step 3: Extract Functionalities

Step 3 shown in Figure 3 is responsible for extracting the annotated functionalities into self-contained *parachute descriptors*. Note that the parachute descriptor is still provider-agnostic as it only contains the function's source code with related data such as class or method dependencies, build scripts, or annotations specified for the next steps. For example, a plain Java function typically cannot be deployed directly to a provider of choice and requires additional post-processing, e. g., AWS Lambda has specific requirements for authoring Java functions. Parachute descriptors can be reused for generating serverless parachutes for other supported providers. The parachute annotations, e. g., proxy configuration directives, are not retained in the extracted functionality, but stored separately in the descriptor. The extraction process is language-specific, hence corresponding extraction plugins are needed to support different programming languages. For example, extraction logic varies due to differences in type support, e. g., strong and weak typing, or the use of different build tools. During extraction, various information is captured in the descriptor, including: (i) endpoint path, (ii) input and output types, also, if needed, with the corresponding type definitions, (iii) class, method, and variable dependencies, and (iv) library dependencies, e. g., a naïve superset of the build script dependencies. The resulting descriptor is self-contained and allows generating provider-specific FaaS deployments without performing the extraction step again.

D. Step 4: Generate Serverless Parachutes

Step 4 shown in Figure 3 is responsible for generating *parachute deployment bundles* that contain extracted functionalities in provider-specific format, e. g., AWS Lambda deployment packages, together with the corresponding deployment models, e. g., modeled by means of AWS CloudFormation templates or using TOSCA [39]–[41] cloud modeling language. In this step, previously created parachute descriptors serve as input for generating the packaging format of a chosen provider. To generate a provider-specific format from the descriptor, the extracted method and its dependencies need to be adapted, e. g., creating correct packages and classes, adding specific import statements, changing method signatures as required by the provider. Moreover, the build script might need to be enriched with provider-specific dependencies. In cases where compilation is required, it needs to be triggered, e. g., to provide an AWS-deployable JAR. Resulting artifacts become a part of a generated deployment model for automating the parachutes deployment. For example, an AWS Serverless Application Model (SAM) template can be generated to automatically deploy all generated AWS Lambda functions. The parachute deployment bundle also includes the generated proxy configuration files and corresponding deployment models to automate the creation of the proxy component. Conditional routing is configured based on the annotations provided in the second step by mapping them to supported underlying router types such as Nginx, Envoy, or custom routers, e. g., relying on Envoy's extensibility mechanisms.

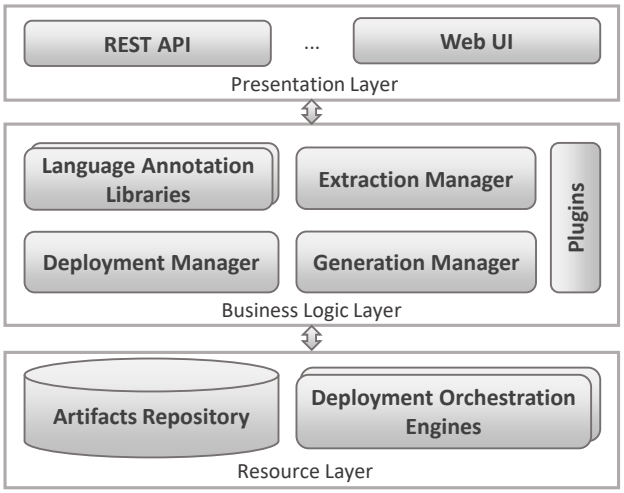

Fig. 4. The architecture of the serverless parachutes framework

E. Steps 5&6: Test, Refine, and Deploy

In Step 5 and Step 6 shown in Figure 3, the generated artifacts can be first tested and refined and then automatically deployed. To avoid repetition, deployment automation is used to simplify the deployment of all generated artifacts, e. g., an AWS Cloud Formation template for proxies and an AWS SAM template for all generated serverless parachutes. If needed, the deployment model also includes required provider's services, e. g., configuration of an API Gateway to expose AWS Lambda functions. In some cases, the generated deployment model might require refinements such as modifying the API specification, changing the application's entry point, or reconfiguring existing proxies instead of using newly-generated ones. While there are various ways to test the deployment, including automated solutions [42], in this paper we do not focus on the deployment testing. Afterwards, the refined parachute deployment bundles can be automatically deployed using a supported deployment orchestration engine.

F. Framework Architecture

An extensible, plugin-based system architecture of the parachutes framework is depicted in Figure 4. The presentation layer enables user interaction by means of an REST API and/or a Web UI. The business logic layer comprises four major

components implementing and orchestrating the introduced method steps while supporting the extensibility through the corresponding plugins. *Language Annotation Libraries* are provided to developers for annotating the crucial functionalities in application's source code as described in Section III-B. The *Extraction Manager* uses language-specific extraction plugins for analyzing the source code and extracting parachute descriptors as discussed in Section III-C. For generating parachute deployment bundles, as discussed in Section III-D, the *Generation Manager* component utilizes required provider-specific generation plugins, e.g., for AWS Lambda or Microsoft Azure Functions. Finally, the *Deployment Manager* is responsible for handling the automatic deployment of refined parachute bundles as discussed in Section III-E by utilizing respective deployment orchestration engines.

The resource layer consists of a set of supported *Deployment Orchestration Engines*, e. g., Terraform or AWS Cloud Formation, used by the Deployment Manager and an *Artifact Repository* which allows storing the artifacts produced by the method for future reuse, e. g., to generate parachute deployment bundles for multiple providers.

IV. Prototypical Validation

As a proof of technical feasibility, we prototypically implemented the framework architecture described in Section III-F using Java [43]. Therefore, the prototype provides an implementation of the three manager components, i. e., Extraction, Deployment and Generation Manager, to conduct and automate the different steps of the parachute method presented in Section III. Furthermore, a language annotation library and an Extraction Manager plugin for applying our method to Java applications, as well as corresponding Generation and Deployment Manager plugins for AWS cloud provider are implemented. The prototype exposes its functionality over a REST API which is implemented using Jersey, an implementation of the JAX-RS specification. An OpenAPI specification of the implemented REST API is generated using respective Swagger tools, and a web-based visualization is accessible through the prototype's interface.

To enable the application of our parachute method, developers need to import the provided Java language annotation library into their Java project for annotating desired functionalities. Afterwards, the prototype can be used for extracting parachutes from the annotated source code. Thus, a developer needs to provide a link to the repository containing the source code, which in our prototype's case is done through providing a GitHub URL. The respective extraction plugin clones the repository and analyzes the source code. All found parachute functionalities are extracted together with their dependencies and meta-data. More specifically, all class and method dependencies including the input and output types, inner classes and methods are extracted together with the functionality and stored in a parachute descriptor file. Furthermore, the build scripts are extracted and included in the descriptor too. The resulting parachute descriptor is serialized using a JSON format which is then used as an input by the respective AWS generation plugin. Java functions for AWS Lambda must conform to certain requirements, e. g., implement *RequestHandler* interface, which requires importing specific AWS packages and modifying the build script's dependencies, e. g., a Maven pom file's dependency list. The AWS generation plugin creates AWS Lambda packages for every extracted parachute by creating a package with all related classes and build scripts, and running the build. Afterwards, an AWS Serverless Application Model (SAM) template is generated for automated deployment of all generated AWS Lambdas. In addition, an AWS Cloud Formation Template is generated for creating a proxy component together with custom configurations based on the provided annotations. As an example of a routing condition, we use serverless parachutes as failover routes when the main application is not available. To implement this routing capability, we use Nginx web server and its configuration of backup routes. Resulting templates are ready for testing, refining, and deployment. We use a file system as an artifact repository for reusing created artifacts within the prototype, i. e., parachute descriptors and deployment bundles.

V. Evaluation

In this section, we describe the evaluation of our approach. First, we present the underlying evaluation methodology applied, followed by a description of the experimental setup, and, finally, a discussion of the evaluation results.

A. Evaluation Methodology and Experimental Setup

The focus of the evaluation is to empirically analyze the performance variation when introducing serverless parachutes compared to the original application deployment. Based on the running example described in Section II-C, for evaluation we use a functionality that generates an invoice in PDF format by taking a JSON object containing the order data as an input. This functionality is implemented as part of an example Java web application[1] and is exposed via a REST API. The goal of evaluation is to identify the overhead introduced by adding a custom proxy component, and analyze the exceptional path activation time in cases when the original functionality becomes unavailable and the parachute takes over the processing responsibility. To achieve this, we measure variations in response times, which are perceived by a user calling a function based on the six scenarios discussed below.

In an *application-plain* scenario, the function is provided as part of the example Java web application that is deployed to an Apache Tomcat version 8.5 to serve incoming requests through its REST API, without applying the serverless parachutes method. In the second scenario called *application-proxy*, a proxy in the form of an Nginx server with a simple reverse proxy configuration is introduced to measure the response time for the plain example application if requests are routed through a reverse proxy, but without applying the serverless parachutes method. The third scenario, *application-failover-proxy*, measures the response time for the plain example application

[1] https://github.com/v-yussupov/parachutesmethod-exampleapp

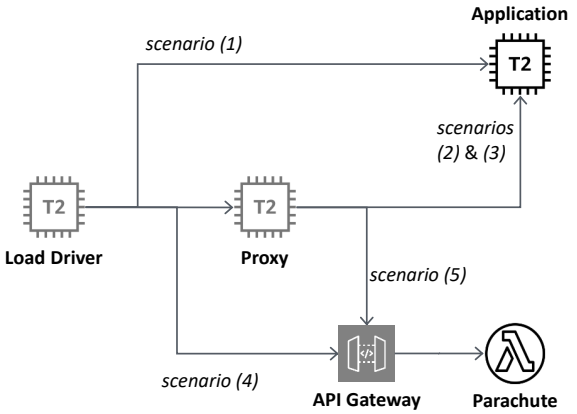

Fig. 5. Experimental setup and evaluation scenarios in AWS infrastructure, using EC2 and Lambda services

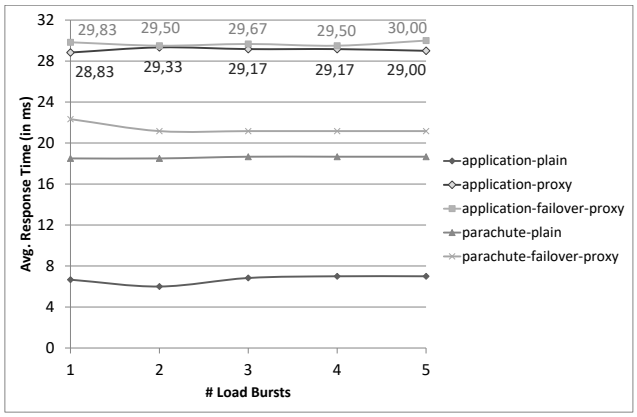

Fig. 6. Average response time in milliseconds (ms) for the load bursts of all scenarios.

accessed via the custom proxy introduced by our method (cf. Section III-D), which is configured to support exceptional cases rerouting for the serverless parachutes method based on an Nginx server. The required custom proxy configuration is generated as a part of the parachute deployment bundle. In the fourth scenario called *parachute-plain*, the response times are measured for the direct communication with the functionality extracted and deployed as a serverless parachute. The extraction, generation, and deployment to AWS Lambda are performed using the prototype, with manual refinement of the endpoints information and API Gateway requests mapping. In the fifth scenario, *parachute-failover-proxy*, we measure the response time for the parachute via the introduced custom proxy configured to support rerouting to serverless parachutes.

In the sixth scenario, *exceptional-path-activation*, we evaluate the times needed to switch from the regular path to exceptional path and vice versa in cases when the application is not available. In this scenario, we measure the response times during the following steps: (i) the functionality is accessed via the regular path (cf. *application-failover-proxy* scenario), (ii) after four minutes, the failover is triggered by stopping the Tomcat to route incoming requests to the parachute (cf. *parachute-failover-proxy* scenario), (iii) after additional four minutes, the Tomcat is started again to restore the regular path. The response times are analyzed to identify the average activation time needed for parachutes to take over. In addition, we analyze the average deactivation time needed to return to the regular path when the application becomes available again.

The experimental environment is set up in the AWS cloud and comprises three AWS EC2 t2.micro (1 vCPU, 1 Gb RAM) instances and one AWS Lambda instance (512 Mb memory) exposed by means of the AWS API Gateway. EC2 instances host: (i) Apache JMeter[2] as a load driver, (ii) Nginx as a proxy (either a reverse proxy or a failover proxy configuration), and (iii) example Java web application deployed to Apache Tomcat. Figure 5 depicts the experimental setup and communication paths for these five scenarios using a set of official AWS Architecture Icons[3]. For each of the six scenarios, a workload consisting of random 2KB JSON HTTP requests is generated. The workload is distributed among a *warm-up phase* (w) with 2000 requests followed by an *experimental phase* comprising a set of 31000 requests sent in five load bursts (i) for each user according to the following function:

$$m(i) = w + \sum_{i=1}^{5} 2^{i-1} \cdot 1000 \mid w = 2000$$

Where $m(i)$ reflects the total number of requests send until the i-th load burst with an increase of requests to the power of two across the load burst.

For the failover scenario requests are sent constantly by five concurrent users without any load bursts after the warmup-phase to measure the failover time when the function at the Tomcat becomes unavailable and Nginx redirects all incoming requests to the parachute deployed at AWS Lambda.

To conduct the workload for each of the defined evaluation scenarios, we use Apache JMeter version 5.1.1. We created a JMeter test plan for each of the introduced six scenarios which concurrently sends the above defined workload for five concurrent users to the endpoint of the function. To alleviate the effect of outliers in the experimental results, we execute six rounds of each scenario and calculate the average response time for each load burst. All JMeter test plans and evaluation results are available online[4].

B. Experimental Results

Figure 6 demonstrates the evaluation results for the first five scenarios. The total amount of requests sent when evaluating six described scenarios is around 5.9 million requests. For the *application-plain* scenario the response time is approximately 7ms, while the average response time for a direct AWS Lambda communication in the *parachute-plain* scenario is approximately 19ms. The increase in response time of approximately 12ms for the *parachute-plain* scenario might be due

[2]Apache JMeter: http://jmeter.apache.org/

[3]https://aws.amazon.com/architecture/icons

[4]https://github.com/v-yussupov/parachutesmethod-evaluation

to the fact that the AWS Lambda is invoked via the AWS API Gateway, which needs to, e. g., map requests and responses or convert the response into a binary format. When introducing the failover proxy in the *parachute-failover-proxy* scenario, the response time slightly increases to approximately 21ms. This shows that introducing a failover proxy before a parachute does only result in an overhead of approximately 3ms. Almost identical average response times with approximately 30ms are captured for the *application-proxy* and *application-failover-proxy* scenarios. The difference between communication with the application directly or via the proxy is consistent across all bursts and is approximately 23ms.

For the sixth scenario, we evaluated the exceptional path's activation and deactivation time. The average time required for activating the exceptional path is approximately 3.3 seconds, which is strongly influenced by the cold start problem [3], i. e., the initialization time required for spinning up the first instance of a function. Within our experiments, it took only 4-5 requests until the response time from the activated parachute reduced significantly towards the average response times measured in the other five scenarios. On the opposite, the deactivation time, i. e., switching back from the parachute to the original application, was in average 380ms. Furthermore, the failover performance remained constant over all executed six evaluation rounds, making our parachute method applicable to events of high and unexpected loads.

In general, the evaluation results indicate that adding the failover proxy component, which reroutes traffic between the application and AWS Lambda function, introduces an overhead that has to be considered when applying the serverless parachutes method. Therefore, in future work we plan to optimize our method to decrease the performance overhead as well as conduct an extended evaluation where we measure the influence of different types of extracted functions, varying request sizes, proxy configurations, and other cloud providers.

VI. Conclusion and Future Work

In this work we presented a method for preparing chosen functionalities of existing or developed applications to withstand unexpected loads caused by exceptional cases. Our method leverages the advantages of the FaaS cloud service model by generating and automatically deploying FaaS versions of the chosen functionalities, while keeping the architecture of the original application untouched and adapting the overall system architecture with generated and configured proxy components. Chosen functionalities are annotated on the level of source code and then automatically extracted into parachute descriptors, which are used to generate deployable parachutes for chosen cloud service providers. As a proof of concept, we implemented a prototype supporting automatic extraction, generation, and deployment of serverless parachutes for Java projects and AWS cloud provider. Furthermore, we evaluated our method by conducting a set of experiments and estimating the overhead introduced by the generated proxy components and measuring the exceptional path activation/deactivation times. The overhead for a generated proxy component shown in our preliminary evaluation results need to be taken into consideration. The extracted parachutes demonstrate a stable response time and nearly 100% success rate when the original application becomes unavailable. The evaluation results demonstrate that our method is applicable as a backup strategy for cases when unexpected, high loads target crucial application functionalities.

Apart from using it as a backup strategy, the serverless parachutes method can be used in alternative application scenarios. One obvious example is an intermediate step before strangulating the functionality, e. g., blue-green or canary deployments. In case a FaaS deployment of the functionality is eventually favored over the original functionality, the method can be extended with additional steps, e. g., analyze the dependencies and strip off the annotated functionality. Interestingly, the method can be applied to both, existing and newly-developed applications. Another application scenario is a simplification of a loosely-coupled architecture development following a monolith-first approach [44]. Here, the to-be-moved functionalities can be annotated and tested in parachute deployments without affecting the monolith's development.

In future work, we plan to perform an extended evaluation, which will include different payload sizes, programming languages, cloud providers, and proxy settings. We intend to further optimize the proxy generation and configuration as well as to introduce support for different proxy types, e. g., using the extensibility features of Envoy. Furthermore, we plan to enrich our prototype with additional plugins for extraction and generation steps to support other programming languages and cloud providers, including open source FaaS platforms as well as to extend the supported annotation types and to investigate other ways of applying our method, e. g., using the ideas of Aspect-Oriented Programming [45]. One optimization strategy that we intend to explore is reconfiguration of existing components in the application's deployment instead of introducing new proxies, e. g., application load balancers can be reconfigured to avoid additional overhead.

In addition, there are several interesting research questions, which can help extending and enhancing the parachutes method. Firstly, we plan to investigate how the parachutes method can be extended to support strong code mobility [33], and, in particular, remote cloning mechanism. With remote cloning, the chosen functionality can be dynamically extracted from the source application at runtime, i. e., the application is dynamically adapted to scale only its certain parts. Another question is to provide developers with the guidelines and recommendations on which functionalities are suitable for parachuting them, e. g., by means of profiling functions using metrics like the number of dependencies or presence of database operations in the code.

Acknowledgment

This work is partially funded by the European Union's Horizon 2020 research and innovation project *RADON* (825040). We would also like to thank the three anonymous reviewers, whose insightful feedback helped to improve this paper.

REFERENCES

[1] P. M. Mell and T. Grance, "Sp 800-145. the NIST definition of cloud computing," Gaithersburg, MD, United States, Tech. Rep., 2011.

[2] Y. Izrailevsky and C. Bell, "Cloud reliability," *IEEE Cloud Computing*, vol. 5, no. 3, pp. 39–44, 2018.

[3] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski *et al.*, "Serverless computing: Current trends and open problems," in *Research Advances in Cloud Computing*. Springer, 2017, pp. 1–20.

[4] Cloud Native Computing Foundation (CNCF). (2018, September) CNCF Serverless Whitepaper v1.0. [Online]. Available: https://github.com/cncf/wg-serverless/tree/master/whitepapers/serverless-overview

[5] A. Eivy, "Be wary of the economics of" serverless" cloud computing," *IEEE Cloud Computing*, vol. 4, no. 2, pp. 6–12, 2017.

[6] Amazon Web Services, Inc. (2019, February) AWS Lambda. [Online]. Available: https://aws.amazon.com/lambda

[7] Microsoft. (2019, February) Microsoft Azure Functions. [Online]. Available: https://azure.microsoft.com/en-us/services/functions

[8] N. Kulkarni and V. Dwivedi, "The role of service granularity in a successful soa realization a case study," in *Services-Part I, 2008. IEEE Congress on*. IEEE, 2008, pp. 423–430.

[9] O. Mustafa and J. M. Gómez, "Optimizing economics of microservices by planning for granularity level," *Experience Report*, 2017.

[10] Y.-W. Kwon and E. Tilevich, "Cloud refactoring: automated transitioning to cloud-based services," *Automated Software Engineering*, vol. 21, no. 3, pp. 345–372, 2014.

[11] W. Ye, A. I. Khan, and E. A. Kendall, "Distributed network file storage for a serverless (p2p) network," in *The 11th IEEE International Conference on Networks, 2003. ICON2003.* IEEE, 2003, pp. 343–347.

[12] C. C. Tan, B. Sheng, and Q. Li, "Secure and serverless rfid authentication and search protocols," *IEEE Transactions on Wireless Communications*, vol. 7, no. 4, pp. 1400–1407, 2008.

[13] G. C. Fox, V. Ishakian, V. Muthusamy, and A. Slominski, "Status of serverless computing and function-as-a-service (faas) in industry and research," *arXiv preprint arXiv:1708.08028*, 2017.

[14] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," *arXiv preprint arXiv:1812.03651*, 2018.

[15] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, "Serverless computing: An investigation of factors influencing microservice performance," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2018, pp. 159–169.

[16] R. N. Charette, "Why software fails [software failure]," *Ieee Spectrum*, vol. 42, no. 9, pp. 42–49, 2005.

[17] P. Garraghan, R. Yang, Z. Wen, A. Romanovsky, J. Xu, R. Buyya, and R. Ranjan, "Emergent failures: Rethinking cloud reliability at scale," *IEEE Cloud Computing*, vol. 5, no. 5, pp. 12–21, 2018.

[18] B. Maurer, "Fail at scale," *Queue*, vol. 13, no. 8, p. 30, 2015.

[19] T. Asghar, S. Rasool, M. Iqbal, Z. ul Qayyum, A. N. Mian, and G. Ubakanma, "Feasibility of serverless cloud services for disaster management information systems," in *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 2018, pp. 1054–1057.

[20] T. Guo, U. Sharma, P. Shenoy, T. Wood, and S. Sahu, "Cost-aware cloud bursting for enterprise applications," *ACM Transactions on Internet Technology (TOIT)*, vol. 13, no. 3, p. 10, 2014.

[21] K. Coleman, F. Esposito, and R. Charney, "Speeding up children reunification in disaster scenarios via serverless computing," in *Proceedings of the 2nd International Workshop on Serverless Computing*. ACM, 2017, pp. 5–5.

[22] O. Mustafa, J. M. Gómez, M. Hamed, and H. Pargmann, "Granmicro: A black-box based approach for optimizing microservices based applications," in *From Science to Society*. Springer, 2018, pp. 283–294.

[23] S. Newman, *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.", 2015.

[24] D. Taibi, V. Lenarduzzi, and C. Pahl, "Processes, motivations, and issues for migrating to microservices architectures: an empirical investigation," *IEEE Cloud Computing*, no. 5, pp. 22–32, 2017.

[25] P. Di Francesco, P. Lago, and I. Malavolta, "Migrating towards microservice architectures: an industrial survey," in *2018 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 2018.

[26] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil, "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," in *Computing Colombian Conference (10CCC), 2015 10th*. IEEE, 2015, pp. 583–590.

[27] R. Khadka, G. Reijnders, A. Saeidi, S. Jansen, and J. Hage, "A method engineering based legacy to soa migration method," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2011, pp. 163–172.

[28] S. Frey and W. Hasselbring, "The cloudmig approach: Model-based migration of software systems to cloud-optimized applications," *International Journal on Advances in Software*, vol. 4, no. 3 and 4, pp. 342–353, 2011.

[29] J. Spillner, "Transformation of python applications into function-as-a-service deployments," *arXiv preprint arXiv:1705.08169*, 2017.

[30] J. Spillner and S. Dorodko, "Java code analysis and transformation into aws lambda functions," *arXiv preprint arXiv:1702.05510*, 2017.

[31] K. J. P. G. Perera and I. Perera, "Thearchitect: A serverless-microservices based high-level architecture generation tool," in *2018 IEEE/ACIS 17th International Conference on Computer and Information Science (ICIS)*, June 2018, pp. 204–210.

[32] M. Fowler. (2004, June) StranglerApplication. [Online]. Available: https://www.martinfowler.com/bliki/StranglerApplication.html

[33] A. Fuggetta, G. P. Picco, and G. Vigna, "Understanding code mobility," *IEEE Transactions on software engineering*, vol. 24, no. 5, pp. 342–361, 1998.

[34] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with openlambda," *Elastic*, vol. 60, p. 80, 2016.

[35] X. Geng, O. Ma, Y. Pei, Z. Xu, W. Zeng, and J. Zou, "Research on early warning system of power network overloading under serverless architecture," in *2018 2nd IEEE Conference on Energy Internet and Energy System Integration (EI2)*. IEEE, 2018, pp. 1–6.

[36] L. Baresi, D. F. Mendonça, and M. Garriga, "Empowering low-latency applications through a serverless edge computing architecture," in *European Conference on Service-Oriented and Cloud Computing*. Springer, 2017, pp. 196–210.

[37] M. Malawski, A. Gajek, A. Zima, B. Balis, and K. Figiela, "Serverless execution of scientific workflows: Experiments with hyperflow, aws lambda and google cloud functions," *Future Generation Computer Systems*, 2017.

[38] P. Dziurzanski, J. Swan, and L. Soares Indrusiak, "Value-based manufacturing optimisation in serverless clouds for industry 4.0," in *Proceedings of the Genetic and Evolutionary Computation Conference*. York, 2018.

[39] U. Breitenbücher *et al.*, "Combining declarative and imperative cloud application provisioning based on tosca," in *Proceedings of the IEEE International Conference on Cloud Engineering (IEEE IC2E 2014)*. IEEE Computer Society, March 2014, pp. 87–96.

[40] M. Wurster, U. Breitenbücher, K. Képes, F. Leymann, and V. Yussupov, "Modeling and Automated Deployment of Serverless Applications using TOSCA," in *Proceedings of the IEEE 11th International Conference on Service-Oriented Computing and Applications (SOCA)*. IEEE Computer Society, 2018, pp. 73—80.

[41] V. Yussupov, M. Falkenthal, O. Kopp, F. Leymann, and M. Zimmermann, "Secure Collaborative Development of Cloud Application Deployment Models," in *Proceedings of The 12th International Conference on Emerging Security Information, Systems and Technologies (SECURWARE 2018)*. Xpert Publishing Services, 2018, pp. 48–57.

[42] M. Wurster, U. Breitenbücher, O. Kopp, and F. Leymann, "Modeling and Automated Execution of Application Deployment Tests," in *Proceedings of the IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC)*. IEEE Computer Society, 2018, pp. 171–180.

[43] (2019, May) Prototypical implementation of the Serverless Parachutes Framework. [Online]. Available: https://github.com/v-yussupov/parachutesmethod-framework/releases/tag/v0.1.0

[44] M. Fowler. (2015, June) StranglerApplication. [Online]. Available: https://www.martinfowler.com/bliki/MonolithFirst.html

[45] G. Kiczales and M. Mezini, "Aspect-oriented programming and modular reasoning," in *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* IEEE, 2005, pp. 49–58.