Hochschule Reutlingen
Reutlingen University

Parallel and Distributed Computing Group
Department of Computer Science
Reutlingen University

AUTOGENIC: Automated Generation of Self-configuring Microservices

Stefan Kehrer, Wolfgang Blochinger
Reutlingen University
{stefan.kehrer, wolfgang.blochinger}@reutlingen-university.de

```
@conference{KEHRER2018_CLOSER,
       author={Stefan Kehrer and Wolfgang Blochinger},
       title={AUTOGENIC: Automated Generation of Self-configuring Microservices},
       booktitle={Proceedings of the 8th International Conference on Cloud
       Computing and Services Science - Volume 1: CLOSER,},
       year={2018},
       pages={35-46},
       publisher={SciTePress},
       organization={INSTICC},
       doi={10.5220/0006659800350046},
       isbn={978-989-758-295-0},
}
```

The full version of this publication has been presented at CLOSER 2018.
http://closer.scitevents.org/?y=2018

Original Publication Reference:
http://www.scitepress.org/PublicationsDetail.aspx?ID=cvCeJMRXnsM=&t=1

AUTOGENIC: Automated Generation of Self-configuring Microservices

Stefan Kehrer[1] and Wolfgang Blochinger[1]
[1] *Department of Computer Science, Reutlingen University, Alteburgstr. 150, 72762 Reutlingen, Germany*
{*stefan.kehrer, wolfgang.blochinger*}*@reutlingen-university.de*

Keywords: Microservices, DevOps, Container Virtualization, Configuration, Service Registry, TOSCA, Docker, Consul

Abstract: The state of the art proposes the microservices architectural style to build applications. Additionally, container virtualization and container management systems evolved into the perfect fit for developing, deploying, and operating microservices in line with the DevOps paradigm. Container virtualization facilitates deployment by ensuring independence from the runtime environment. However, microservices store their configuration in the environment. Therefore, software developers have to wire their microservice implementation with technologies provided by the target runtime environment such as configuration stores and service registries. These technological dependencies counteract the portability benefit of using container virtualization. In this paper, we present AUTOGENIC - a model-based approach to assist software developers in building microservices as self-configuring containers without being bound to operational technologies. We provide developers with a simple configuration model to specify configuration operations of containers and automatically generate a self-configuring microservice tailored for the targeted runtime environment. Our approach is supported by a method, which describes the steps to automate the generation of self-configuring microservices. Additionally, we present and evaluate a prototype, which leverages the emerging TOSCA standard.

1 INTRODUCTION

Today's business environment requires fast software release cycles. To address this issue, continuous delivery and DevOps aim at bridging the gap between development and operations by employing automation and self-service tools. Microservices are an evolving architectural style for building and releasing software in line with the DevOps paradigm (Balalaie et al., 2016; Pahl and Jamshidi, 2016). Microservices are autonomous and independently deployable (Newman, 2015).

Unfortunately, the autonomous nature of microservices challenges their development: More and more operational aspects are transferred into the responsibility of software developers - or how Amazon calls it: "you build it, you run it" (O'Hanlon, 2006). This is also enabled by technological advances such as container virtualization (Kratzke and Quint, 2017; Pahl and Jamshidi, 2016): Microservices are commonly built as a set of containers, which provide a portable means to deploy microservices on state of the art container management systems such as Marathon[1] on Apache Mesos[2], Kubernetes[3], or Docker Swarm[4].

In line with this trend, software developers have to implement their microservices including operational behavior. Every container that is part of the microservice has to be configured with specific runtime parameters as well as endpoint information to interact with other containers. This configuration of containers might be applied during the deployment of a microservice. However, in a dynamic environment such as the cloud, dynamic updates of runtime parameters might be required. Furthermore, endpoint information will likely change during runtime, e.g., if a container has to be restarted. Thus, microservices store their configuration in the environment[5]. This means that configuration stores are used to store required runtime parameters and service registries are used to find other containers. Following this approach, software developers have to wire their microservice implementation with technologies provided by the runtime environment. Besides adding more complexity, this leads to heterogeneous implementations of configuration management. Moreover, technological de-

[1] https://mesosphere.github.io/marathon.

[2] https://mesos.apache.org.

[3] https://kubernetes.io.

[4] https://github.com/docker/swarm.

[5] https://12factor.net/config.

pendencies on configuration stores and service registries provided by the runtime environment decrease the portability benefit inherent to containers.

To address the aforementioned challenges, we present a novel approach called *AUTOmated GENeration of self-configuring mICroservices (*AUTOGENIC*)*. Our model-based approach enables software developers to specify the configuration operations of containers with a configuration model. Based on this model, we transform a supplied microservice into a self-configuring microservice by automatically adding runtime behavior to its containers on a technical level. As a result, configuration is managed by each container and thus accomplished in a decentralized manner. This transformation is provided as a service to developers and thus decouples environment-specific technologies from application development. Our approach streamlines the cooperation of developers and operations personnel by providing an abstraction layer between both groups, which basically implements the separation of concerns principle in the DevOps context. In particular, we present the following contributions:

- We introduce the AUTOGENIC approach to assist software developers in creating self-configuring microservices.
- We provide the AUTOGENIC method, which describes the steps of generating self-configuring microservices on a conceptual level.
- We present an implemented prototype, which automates the AUTOGENIC method based on the TOSCA standard and state of the art technologies.

The paper is structured as follows. In Section 2, we describe microservices in general and motivate our work. Section 3 gives an overview of the general concepts of the AUTOGENIC approach. In Section 4, we discuss the AUTOGENIC method, which describes the required steps to automate the generation of self-configuring microservices. Further, we present an implemented prototype in Section 5 and evaluate this prototype in Section 6. In Section 7, we review related work. Finally, Section 8 concludes this paper and describes future work.

2 STATE OF THE ART AND MOTIVATION

In this section, we describe the state of the art and present an exemplary microservice to motivate our work.

2.1 Microservices

A microservice is built around a business capability and implements the user interface, storage, and any external collaborations required (Lewis and Fowler, 2014). Thus, each microservice is a *broad-stack implementation of software* for a specific business capability (Lewis and Fowler, 2014). Microservices combine concepts from distributed systems and service-oriented architecture leading to several benefits (Newman, 2015). For instance, microservices can be implemented with different technologies enabling a best-of-breed approach. Thus, new technologies can be adopted and old technologies can be replaced much faster. Composing a system out of many small services also provides benefits for deployment and management: It allows to deploy and scale every microservice independently (Leymann et al., 2017). Typically, software containers are used to package and deploy microservice components (Pahl et al., 2017). A topology model or template, which describes the containers a microservice is composed of and their relationships, enables automated deployment (Kehrer and Blochinger, 2018).

However, the benefits of microservices come with the cost of operational complexity (Fowler, 2017). The autonomous nature inherent to microservices requires application developers to take responsibility for operational aspects such as dynamic configuration (Kookarinrat and Temtanapat, 2016). To this end, the *Twelve-Factor App*[6] principles propose to store these information in the runtime environment. Technologies such as configuration stores and service registries are used to store configuration values and enable dynamic bindings among containers. Employing technologies like Consul[7], Etcd[8], or Zookeeper[9] is a common practice for developing microservices (Toffetti et al., 2017). They provide a scalable medium to store configuration information.

2.2 Motivation

In this section, we introduce a microservice, which is used as motivating example for our work. The topology of this microservice is composed of four containers interacting with each other (cf. Figure 1): The *wordpress* container provides an Apache HTTP server running a WordPress installation. The *mysql* container runs a MySQL database. To answer user requests, the *wordpress* container connects to the *mysql*

[6]https://12factor.net.
[7]https://www.consul.io.
[8]https://github.com/coreos/etcd.
[9]https://zookeeper.apache.org.

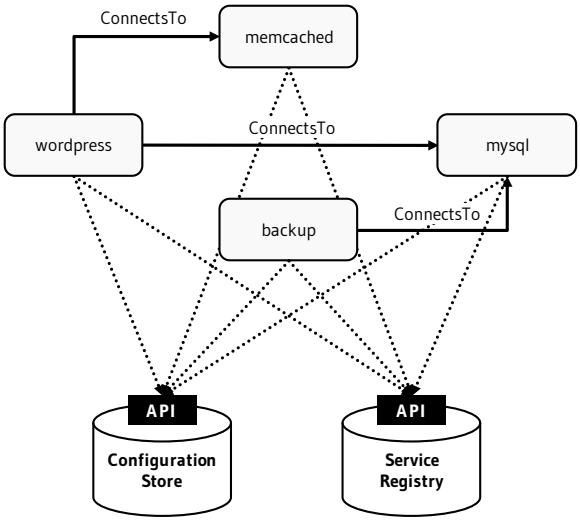

Figure 1: Exemplary microservice storing its configuration in the runtime environment

container and retrieves data stored in the relational database. Frequently requested results are cached in the *memcached* container, which runs a Memcached[10] installation. Memcached is an in-memory object caching system. The *memcached* container is queried by the *wordpress* container before sending a read request to the *mysql* container. Additionally, a separate *backup* container periodically stores backups of the MySQL database by connecting to the corresponding container.

For configuration purposes, every container of the formerly described microservice requires its runtime parameters and endpoint information to interact with other containers in the topology. To access their runtime parameters, the containers connect to a configuration store provided by the runtime environment. Similarly, every container connects to a service registry to access endpoint information of other containers (cf. Figure 1). Whenever a runtime parameter or endpoint information changes in the environment, a container itself is responsible for reacting to this change. This results in software developers having to wire their implementations with operational technologies provided by the runtime environment.

We identified several problems with this approach: (1) APIs of the configuration store and the service registry have to be used by software developers. Every time the operations personnel decides to choose another technology, software developers have to be instructed and existing microservice implementations have to be modified. (2) Storing endpoint information of containers belonging to a microservice in a central service registry may lead to conflicts with other deployments and breaks the microservice paradigm, e.g., if another service requester receives the endpoint information of our MySQL database. This information should be kept private and not exposed to other microservices (O'Hanlon, 2006; Lewis and Fowler, 2014). (3) Moreover, portability is limited, i.e., microservices cannot be deployed on a runtime environment that does not provide the required technologies.

In general, software developers are confronted with a lot of often changing technologies to enable dynamic configuration. Technological dependencies on specific configuration stores or service registries counteract the portability benefit of using container virtualization. New solutions are required, which assist software developers in implementing dynamic configuration for their microservices.

3 AUTOMATED GENERATION OF SELF-CONFIGURING MICROSERVICES

We propose *AUTOmated GENeration of self-configuring mICroservices (*AUTOGENIC*)* to assist software developers in building dynamically configuring microservices. We aim at providing a simple means for software developers to take responsibility for operational aspects of their microservice in line with the "you build it, you run it" principle. We identified two fundamental design guidelines for such an approach: (1) Software developers have to be able to control the configuration of containers belonging to a microservice. (2) Technological details should be hidden from software developers to enable portability and operational flexibility with respect to the runtime environment and tool support.

Basically, AUTOGENIC is a model-based approach to decouple the development of microservices from environment-specific technologies provided by operations personnel. Software developers simply specify configuration operations of their microservice in a *configuration model*, i.e., without considering the specific technologies present in the runtime environment. Based on this model, the required runtime behavior can be automatically derived and mapped to operational technologies. This enables the design of a self-service tool for software developers to automatically transform their microservice into a self-configuring microservice tailored for the targeted runtime environment (cf. Figure 2).

Microservices are constructed as independently deployable units. Thus, we assume some kind of *ser-*

[10]https://memcached.org.

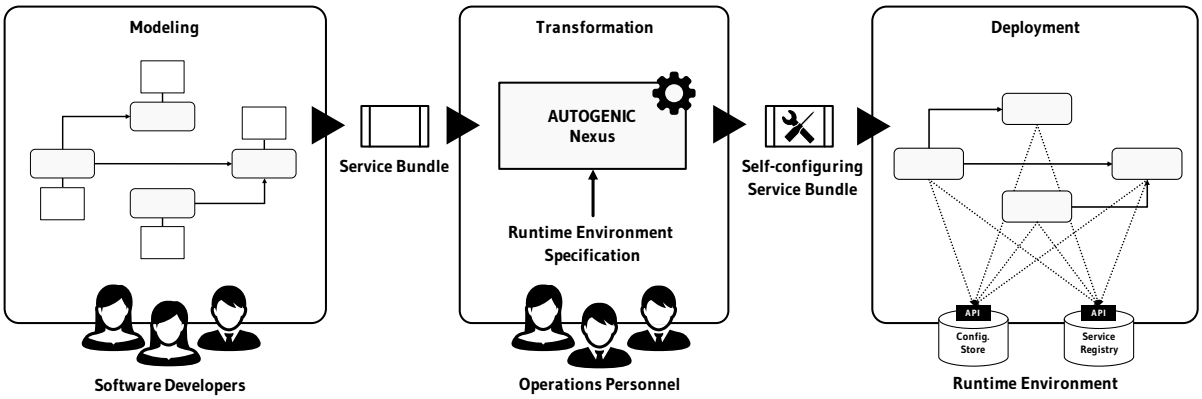

Figure 2: Overview of the AUTOGENIC approach

vice bundle, which contains all the required artifacts to deploy a microservice. An important part of the service bundle is the topology model describing the topology of containers and related artifacts (e.g., container images) (Kehrer and Blochinger, 2018). The topology model contains all information required to automatically deploy a corresponding microservice to a runtime environment. However, besides specifying the containers and their relationships for deployment purposes, developers also have to consider the dynamic configuration of these containers during runtime (cf. Section 2.1).

To specify the configuration requirements, we utilize the existing topology model, which is part of every service bundle. Following a model-based approach, we enable developers to annotate each container specified in the topology model with a configuration model. Figure 3 shows two containers, which are part of a topology model, each annotated with a configuration model. The configuration model contains one or more *configuration operations*. These configuration operations are defined by a name and specify an implementation artifact as well as inputs. The implementation artifact refers to an executable artifact in the container (e.g., a shell script) that must be invoked to execute the configuration operation on a technical level. The inputs can be defined as key-value pairs, which are passed to the implementation artifact upon execution. In case of our exemplary microservice, a shell script for connecting to the MySQL database might be specified as implementation artifact of the *configure_db* operation attached to the *wordpress* container (cf. Figure 3).

Additionally, we enable the use of *functions* to specify input values for configuration operations. Functions can be used to reference dynamic attribute values of entities in the topology model, e.g., IP addresses of modeled containers. Referring to our exemplary microservice, the *configure_db* operation specifies an input named *mysql_ip* with the function *getIPAddress()* that retrieves the IP address of the *mysql* container (cf. Figure 3).

A core idea of the AUTOGENIC approach is to automatically execute configuration operations whenever their input values change. Since these input values are stored in the runtime environment, a corresponding *event-trigger* has to be registered to this change event in the environment. The callback of this event-trigger is given by the implementation artifact specified for the corresponding configuration operation. This enables reactive configuration and dynamic bindings among containers. A typical example is the *configure_db* operation of the *wordpress* container. Reconfiguration and thus execution of the */configure.sh* script is required whenever the IP address of the *mysql* container changes in the environment.

The topology model enhanced with the proposed configuration model (cf. Figure 3) is packaged into a service bundle and then passed as input to a service that we call AUTOGENIC *Nexus* (cf. Figure 2). The

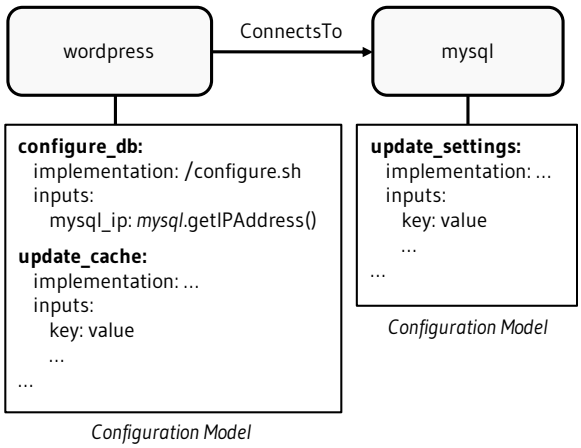

Figure 3: Containers annotated with configuration models

AUTOGENIC Nexus takes a developer-supplied service bundle as input and generates a self-configuring service bundle as output. The transformation applied adds self-configuration mechanisms to each container on a technical level based on the configuration operations specified. The AUTOGENIC *Nexus* is maintained by operations personnel and provided as a self-service tool to software developers. It encapsulates the specifics of the target runtime environment. This might be the access mechanisms of the configuration store and the service registry used in the runtime environment (e.g., APIs) as well as event-dispatching mechanisms to trigger developer-supplied implementation artifacts. The selection of these technologies is an operational decision and thus should be handled transparently to application development. The runtime environment specification has to be considered during the implementation of the AUTOGENIC Nexus.

Implemented once, the AUTOGENIC Nexus provides a self-service tool for developers, which generates self-configuring service bundles targeted to a specific runtime environment without any knowledge on operational technologies employed. The self-configuring service bundle contains all required information to deploy a microservice in an automated manner (cf. Figure 2). This approach ensures the separation of concerns principle in the DevOps context in line with our design guidelines defined above.

Following our model-based approach, service bundles can be developed independently of the runtime environment. This leads to several benefits compared to microservice configuration on programming level, i.e., directly implementing the API of a configuration store or service registry: (1) Different technologies can be used to implement the required configuration behavior depending on the target runtime environment; (2) Developers do not have to build triggers for configuration operations by wiring APIs. Configuration operations are executed automatically whenever their input values change; (3) Logical identifiers of containers are only used in the model and not in the containers themselves leading to higher reusability. Further, these identifiers are private to the topology model of a single microservice and thus cannot be used by other microservices. Note that this is an important requirement, e.g., to prevent direct database access from outside the service (O'Hanlon, 2006).

4 AUTOGENIC METHOD

The AUTOGENIC method specifies the steps to transform an existing service bundle including its configuration models into a self-configuring service bundle. This method describes the transformation performed by the AUTOGENIC Nexus on a conceptual level to guide the runtime-specific implementation by operations personnel. Accordingly, our method describes the transformation independently of (1) the modeling language used for the topology and configuration models, (2) the container format employed for virtualization, (3) operational technologies in the target runtime environment, and (4) event-dispatching mechanisms used to build event-triggers. As a result, our method supports various combinations, which can be found in practice (cf. Section 5). Figure 4 depicts the AUTOGENIC method. We describe its steps in the following.

4.1 Assumptions

This method requires a service bundle that contains a topology model enhanced with configuration models. Moreover, build specifications for each container are assumed to be part of the service bundle.

4.2 Step 1: Scan Topology Model & Build Specifications

We assume that each container specified in the topology model links its configuration model and a build specification. Whereas the configuration model describes the desired configuration behavior, the build specification can be used to derive the current runtime behavior of the container. In this step, configuration models and container build specifications are scanned to derive a set of Transformation Requirements (TR). TRs describe the requirements that have to be addressed during the transformation and are provided as input to the next steps. Scanning the configuration models leads to the following TRs:

- A *StoreKeyValueRequirement* describes a key-value pair, which is used as input for a specific configuration operation. This key-value pair has to be stored in the runtime environment during deployment (e.g., by using a configuration store).
- A *KeyWatchRequirement* describes the requirement to watch the value of a specific input key stored in the environment. Whenever the value related to this key changes, the corresponding configuration operation should be executed.
- An *AttributeWatchRequirement* describes the requirement to watch the value of a defined attribute such as the IP address of a specific container. Whenever this value changes the corresponding configuration operation should be executed.

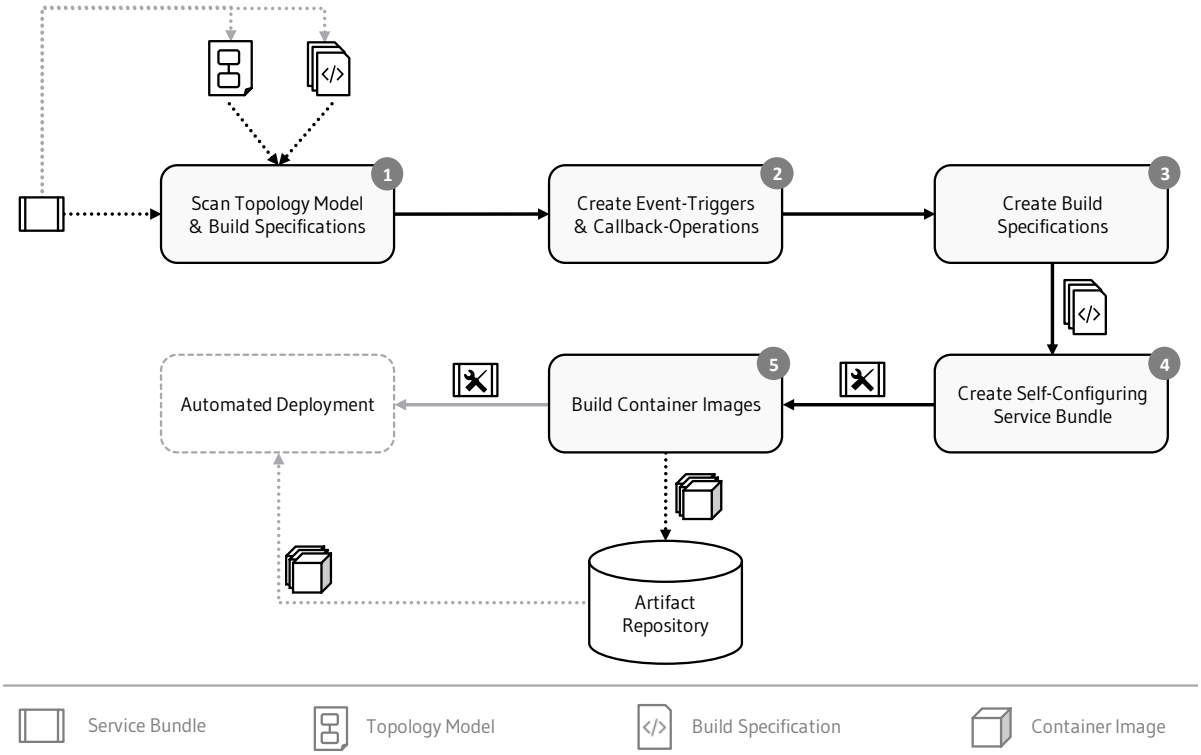

Figure 4: Steps of the AUTOGENIC method

Additionally, the build specifications have to be scanned. On a technical level this is performed by simply recognizing keywords (descriptors) that are defined by the container format employed. Scanning the build specifications leads to the following TR:

- An *EntryPointRequirement* describes the entry-point of a container. This is an executable run at container startup (Turnbull, 2014).

TRs allow the automated construction of a new container image (cf. Section 4.4), which fulfills the same functional requirements as the developer-supplied container image, but additionally contains self-configuration mechanisms.

4.3 Step 2: Create Event-Triggers & Callback-Operations

In this step, the TRs derived have to be addressed. Therefore, implementation artifacts provided by developers have to be bound as callbacks to change events in the environment. Environment-specific event-dispatching mechanisms are employed for this purpose. At the same time, functional aspects of a developer-supplied container should be retained.

StoreKeyValueRequirements are addressed by an initial setup process executed at each container's startup. This setup process stores the required inputs in the environment. After the initial setup process, each container runs the executable captured in its EntryPointRequirement.

KeyWatchRequirements as well as AttributeWatchRequirements have to be met by installing an event-trigger for the corresponding configuration operation, which executes the implementation artifact specified whenever input values change. The implementation of event-triggers depends on the technologies employed in the target runtime environment. This includes mapping the schema of operational data structures as well as defining access methods and protocols for the configuration store and service registry.

This step results in a set of technological artifacts, which ensure dynamic configuration of each container with respect to the target runtime environment. The generated technological artifacts automatically trigger the implementation artifacts supplied by the developer every time a configuration value changes in the environment.

4.4 Step 3: Create Build Specifications

To combine the developer-supplied container image with the technological artifacts generated in Step 2, a new build specification is created for each container specified in the topology model. This build speci-

fication is built on top of the existing build specification that defines the developer-supplied microservice. It basically adds the generated technological artifacts (cf. Section 4.3) and installs required software packages. In this context, a build specification template may be used, which contains the settings derived from the runtime environment specification, e.g., commands to install required software.

4.5 Step 4: Create Self-Configuring Service Bundle

Since configuration operations are now managed by the corresponding container itself, the configuration models are not required for deployment purposes. In this step, a new service bundle is generated, which provides a portable means to deploy the generated self-configuring microservice to the target runtime environment.

4.6 Step 5: Build Container Images

Finally, the container images of the newly generated build specifications captured in the self-configuring service bundle have to be built. Besides creating container images, they have to be pushed to an artifact repository, which can be accessed during deployment.

4.7 Automated Deployment

The generated service bundle provides a means to automatically deploy the generated self-configuring microservice to the target runtime environment. Therefore, container images can be retrieved from the artifact repository specified in the service bundle.

5 AUTOGENIC PROTOTYPE

In this section, we present an AUTOGENIC Nexus prototype. The AUTOGENIC method describes how to transform a service bundle including the configuration models to low-level technical aspects of the target runtime environment. Hence, we have to make four decisions with respect to an implementation: First, we have to specify the modeling language used for topology and configuration models. Possible options are any custom modeling language supporting our assumptions, domain-specific languages of container management systems such as Kubernetes, Marathon, and Docker Swarm as well as the TOSCA standard (OASIS, 2013). Secondly, we have to choose a container format such as Docker, Application Container (appc) Specification[11], or the specification of the Open Container Initiative (OCI)[12]. Thirdly, we have to define the operational technologies of the target runtime environment. Typical examples are Consul, Etcd, ZooKeeper, SkyDNS[13], Eureka[14], and Doozer[15]. Finally, event-dispatching mechanisms are required. Options include specific tooling to access operational technologies as well as ContainerPilot[16].

In this section, we describe a prototype employing the emerging TOSCA standard as modeling language, which also contains a format for service bundles. We rely on the TOSCA standard because it provides a language to specify topology models of microservices in a portable manner and concepts to specify dependencies in the model. The TOSCA concept of Lifecycle Operations already provides us with compatible modeling constructs to specify configuration operations. Further, we employ Docker[17] as container virtualization technology, Consul as configuration store and service registry, and ContainerPilot to build event-triggers. We describe TOSCA and a TOSCA-based service bundle of an exemplary microservice in the following. Moreover, we present an exemplary runtime environment specification. On this basis, we present the implementation of our prototype.

5.1 Topology and Orchestration Specification for Cloud Applications (TOSCA)

The Topology and Orchestration Specification for Cloud Applications (TOSCA) aims at standardizing a modeling language for portable cloud services (OASIS, 2013). Therefore, cloud services are captured as topology graphs modeled in form of a Topology Template. The nodes in the topology are modeled as Node Templates.

Since a Topology Template is an abstract description of a service topology, Deployment Artifacts such as container images (e.g., Docker Images) are linked to Node Templates as depicted in Figure 5. Node Templates also define Lifecycle Operations. These Lifecycle Operations are implemented by Implementation Artifacts such as shell scripts (cf. Figure 5).

Additionally, TOSCA provides a type system that

[11] https://github.com/appc/spec.
[12] https://www.opencontainers.org.
[13] https://github.com/skynetservices/skydns.
[14] https://github.com/Netflix/eureka.
[15] https://github.com/ha/doozerd.
[16] https://github.com/joyent/containerpilot.
[17] https://docker.com.

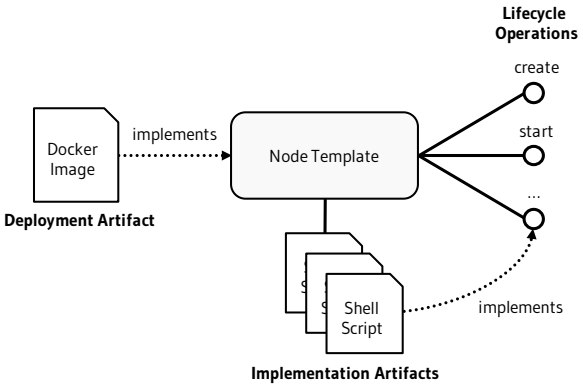

Figure 5: TOSCA Artifacts and Lifecycle Operations

```
1    wordpress:
2      ...
3      contains: [wordpress_build]
4      ...
5      artifacts:
6        wp_image:
7          file: wordpress-custom
8          type:
             ↪  tosca.artifacts.Deployment.Image.Container.Docker
9          repository: custom_repository
10     interfaces:
11       Standard:
12         create:
13           implementation: wp_image
14       Configure:
15         configure_db:
16           implementation: /configure.sh
17           inputs:
18             DB_HOST: { get_attribute: [mysql, ip_address] }
19             DB_USER: myuser
20             DB_PASSWORD: pw
21             DB_NAME: mydb
22         ...
```

Listing 1: wordpress Node Template in YAML

```
1   wordpress_build:
2     ...
3     artifacts:
4       build_spec:
5         file: artifacts/wordpress/Dockerfile
6         type: cst.artifacts.Deployment.BuildSpec.Docker
7         properties:
8           image_name: wordpress-custom
9           repository: custom_repository
10    interfaces:
11      Standard:
12        create:
13          implementation: build_spec
```

Listing 2: wordpress_build Node Template in YAML

allows the definition of custom types such as Node Types or Artifact types. These type definitions and the Topology Template are captured in a so-called Service Template. A TOSCA orchestrator processes a Service Template to instantiate nodes. Modeling a TOSCA-based cloud service results in a self-contained, portable service model called Cloud Service ARchive (CSAR) that can be used to deploy service instances in all TOSCA-compliant environments. The CSAR contains the Service Template and related Deployment Artifacts as well as Implementation Artifacts. In the Simple Profile in YAML V1.0 (OASIS, 2016), TOSCA provides modeling constructs for containers as well as TOSCA Functions. TOSCA Functions allow referencing values of entities in the Topology Template, which have to be resolved during runtime.

5.2 TOSCA-based Service Bundle

In this section, we describe a TOSCA-based service bundle of our exemplary microservice described in Section 2.2. This service bundle will be used as exemplary input for our prototypical implementation. Due to space limitations, we only present representative parts of the service bundle.

We use a CSAR as service bundle, which contains a description of the microservice topology by means of a Topology Template. The Topology Template specifies Node Templates for the containers, namely *wordpress*, *memcached*, *mysql*, and *backup*. Listing 1 shows the Node Template of *wordpress*. It specifies its Deployment Artifact, which is a Docker Image (cf. Listing 1, line 6–9). This Docker Image is provided to the Create Operation to instantiate the node (cf. Listing 1, line 12–13). To specify our configuration models, we append an additional Lifecycle Interface named *Configure* for configuration operations (cf. Listing 1, line 14–22). This Lifecycle Interface provides the information required by the AUTOGENIC Nexus.

The *configure_db* Operation specifies an Implementation Artifact */configure.sh*, which requires four input values. The host of the database is specified with a TOSCA Function (cf. Listing 1, line 18). A TOSCA Function specifies an input value that depends on runtime information. In this case, the IP address of *mysql* is required to connect to the database.

Container images only capture file system changes and thus do not provide information on how they have been created. They are constructed of a set of layers each described by a corresponding build specification such as a Dockerfile. However, the TOSCA standard does not allow the definition of build specifications describing the construction of container images. To resolve this issue, we introduced the concept of *Contained Nodes* (Kehrer and Blochinger, 2018) to model build specifications for each Node Template. Therefore, a *container* Node Template such as *wordpress* links a *contained* Node Template (cf. Listing 1, line 3). The *wordpress_build* Node Template specifies the build specification of the corresponding *wordpress-custom* Docker Image (cf. Listing 2, line 4–9) required to deploy *wordpress*. In this case, the build specification is a Dockerfile.

The containers *memcached*, *mysql*, and *backup* are modeled in an analogous manner and specify their Deployment Artifacts as well as configuration operations as explained above.

5.3 Runtime Environment Specification

The target runtime environment addressed by our prototype is a TOSCA-based container management system from previous work (Kehrer and Blochinger, 2018), which can be used to deploy a TOSCA-based service bundle. We selected Consul to store configuration and endpoint information in the environment, which provides both a key-value store to store configuration data and service discovery mechanisms. The Consul ecosystem provides a rich set of tools to access stored data. To enable self-configuring microservices, we have to additionally select technologies used to bind configuration operations to events. We chose ContainerPilot, which is an open-source project developed by Joyent. ContainerPilot resembles the UNIX concept of process supervision by providing a supervisor middleware for processes running inside a software container. Besides, it provides integration with service discovery tooling, which we apply to bind event-triggers to configuration operations. ContainerPilot is configured by passing a configuration file, which contains the processes to be run. A Docker Registry[18] is employed as artifact repository, i.e., to push and retrieve container images (cf. Figure 4).

5.4 Implementation

In this section, we outline how we implemented our prototype in Java. Therefore, we describe the implementation counterparts of step 1–5 as defined in the AUTOGENIC method (cf. Section 4).

Step 1: A TOSCA Parser loads the TOSCA-based service bundle and transforms the Service Template into an internal object. Our RequirementScanner derives TRs from the Topology Template, namely StoreKeyValueRequirements, KeyWatchRequirements, and AttributeWatchRequirements. Moreover, the RequirementScanner scans the Dockerfiles linked in the Service Template to identify EntryPointRequirements.

Step 2: We employ ContainerPilot version 3.1.1 as process supervisor for each container. A ContainerPilot configuration file is used to create event-triggers for configuration operations. The key-value pairs described by StoreKeyValueRequirements are stored in Consul with an initial setup process executed on container startup. Moreover, the executable captured in an EntryPointRequirement is executed after the initial setup process.

KeyWatchRequirements and AttributeWatchRequirements require the installation of event-triggers. Technically, we register separate background processes in the ContainerPilot configuration file. These background processes run Consul watches with the Consul command line tool, which can be used to get informed whenever a value changes. We use Consul watches to trigger envconsul[19] whenever an input value of a configuration operation changes in Consul. Envconsul then executes the implementation artifact of the corresponding configuration operation and provides the inputs as environment variables. The resulting technological artifacts are a ContainerPilot configuration file and scripts for the initial setup process.

Step 3: To create build specifications, we use a file template for each Dockerfile, which installs a Consul client, envconsul, and ContainerPilot. Additionally, we add the artifacts generated in Step 2. The processing is implemented based on Apache FreeMarker[20], which is an open-source template engine.

Step 4: A new contained Node Template is added to each container Node Template, which is built on top of the developer-supplied contained Node Template and links the generated build specification. Besides, the Deployment Artifacts of the container Node Templates are updated with the name of the new container images. The generated Service Template is added to a newly generated service bundle, which contains all build specifications and technological artifacts required to build the container images.

Step 5: To build container images, we assume a Docker Engine running on the host. We connect to the Docker Engine by using the Docker-Client[21] library developed by Spotify. Docker-Client connects to the Docker Engine through the default UNIX domain socket provided to control Docker-specific functionality. We build the required container images described by the generated build specifications and push them to the artifact repository specified in the Topology Template.

6 EVALUATION

To evaluate our prototype, we employ the formerly described service bundle of our exemplary microser-

[18] https://hub.docker.com/_/registry.

[19] https://github.com/hashicorp/envconsul.

[20] http://freemarker.org.

[21] https://github.com/spotify/docker-client.

vice (cf. Section 5.2). The underlying runtime environment specification is given in Section 5.3. We present two experiments to analyze the overhead resulting from the transformation performed by the AUTOGENIC Nexus prototype.

In the baseline experiment, we build all developer-supplied container images specified in the service bundle and measure the total generation time. We define the total generation time as the accumulated time, which is required to build these container images and to push the generated container images to the artifact repository. In the transformation experiment, we run the prototype to generate a self-configuring service bundle and measure the total transformation time. We define the total transformation time as the elapsed time from the start of the prototype to the point, where all steps of the AUTOGENIC method are successfully completed. This also includes pushing the generated container images to the corresponding artifact repository (cf. Figure 4).

We executed our experiments on a CentOS 7 virtual machine with 2 vCPUs clocked at 2.6 GHz, 4 GB RAM, and 40 GB disk running in our OpenStack-based cloud environment. The virtual machine provides an OpenJDK Runtime Environment 1.8.0 and Docker Engine 1.12.6. For building container images, we rely on the Docker Engine API v1.24. As artifact repository, we run a private Docker Registry v2.6 on localhost. We executed ten independent runs for each experiment and measured the total generation time and the total transformation time, respectively.

In the baseline experiment, we build a single container image for each container. These container images are built based on the build specification specified in the service bundle. However, all container images require base images from the Docker-Hub. The *wordpress* container requires downloading *php:5.6-apache*[22] with 377.7 MB, *memcached* requires *debian:stretch-slim*[23] with 55.24 MB, *mysql* requires *oraclelinux:7-slim*[24] with 117.6 MB, and *backup* requires *python:2.7.14-jessie*[25] with 679.3 MB. To ensure that we measure the total generation time without caching, we cleared the Docker cache and the Docker Registry before every run. In this context, caching of container images relates to the intermediate layers stored by Docker to speed up future build processes. Based on the measurements, we calculated an average total generation time of (882 ± 38) seconds.

In the transformation experiment, we ran our prototype to measure the total transformation time. Therefore, all required container images are built and pushed to the artifact repository. This includes the developer-supplied container images as well as container images generated by the AUTOGENIC Nexus prototype. Again, we cleared the Docker cache and the Docker Registry before every run. Based on the measurements, we calculated an average total transformation time of (1349 ± 16) seconds.

The transformation adds an average overhead in size of 67.8 MB per container image. This is largely related to ContainerPilot and Consul-specific tooling. Note that the container images built in the baseline experiment are not self-configuring. Additional manual effort would be required to enable the same features, thus also leading to larger image sizes.

In summary, the transformation applied by our prototype results in an average overhead of 467 seconds to enable the AUTOGENIC approach. However, we enable software developers to implement their microservices independent of operational technologies, which saves time during development. Moreover, our model-based approach leads to several benefits such as portability of microservice implementations and the separation of concerns for software developers and operations personnel (cf. Section 3).

The overhead measured is basically related to building additional container images, which include the required self-configuration mechanisms. Note that the measurements depend on the size of required and generated container images, the network bandwidth for downloading the required base images, and the location of the artifact repository. Thus, the reported values may be different in a real world scenario. Furthermore, we identified several opportunities to speed up the transformation performance such as building container images concurrently and storing required software packages locally. Obviously, caching techniques offer another opportunity for performance tuning.

7 RELATED WORK

Our approach aims at facilitating the development of self-configuring microservices by introducing an abstraction layer between software developers and operations personnel. Implementing the AUTOGENIC method leads to a self-service tool that enables developers to take responsibility for the dynamic configuration of their microservices independently of the runtime environment. Using self-service tools and automation is a commonly applied approach for supporting DevOps (Hüttermann, 2012).

[22] https://hub.docker.com/_/php.

[23] https://hub.docker.com/_/debian.

[24] https://hub.docker.com/_/oraclelinux.

[25] https://hub.docker.com/_/python.

Microservices require decentralized management and prefer choreography over orchestration (Fowler, 2017; Newman, 2015; Zimmermann, 2017). The authors of (Schermann et al., 2016) state that more research on choreography rather than orchestration is required. Self-configuring microservices are a solution to ensure dynamic configuration without relying on centralized orchestration. Following the AUTOGENIC approach, configuration is managed by each container and thus executed in a decentralized, event-based manner.

Several approaches exist to build microservices with decentralized configuration capabilities. In (Toffetti et al., 2015) and (Toffetti et al., 2017) distributed in-memory key-value stores are employed to communicate changes among components. Whereas this results in a similar technical implementation, our model-based approach contributes to the ease of development of self-configuring microservices. Thus, developers are relieved of the burden of wiring their microservice implementations with operational technologies. In (Stubbs et al., 2015), the authors present a solution to the service discovery problem based on Serf[26]. Their approach proposes an additional *Serfnode* container, which manages a required container instance. In contrast, we add an environment-specific supervisor (e.g., ContainerPilot) directly to an existing container image. Whereas Serfnodes do not require building new container images, they require extra configuration and only solve the service discovery problem. Moreover, the presented solution does not provide the same abstraction level compared to our model-based approach, which uses configuration models to define operational behavior on a higher level.

Microservice chassis[27] such as Spring Cloud[28] might be used to dynamically configure microservices. However, microservice chassis are bound to a specific programming language and are limited to supported operational tooling. Netflix Prana[29] provides a side car for services based on the NetflixOSS[30] ecosystem. This enables the use of Java-based NetflixOSS libraries for microservices written in other programming languages. Registrator[31] enables service discovery features for Docker containers by watching the runtime environment. In comparison, we provide a simple means to software developers and separate the logical definition of configuration operations from their technical implementation. Following this model-based approach enables the use of different technological solutions depending on the target runtime environment.

[26]https://www.serf.io.

[27]http://microservices.io/patterns/microservice-chassis.html.

[28]http://projects.spring.io/spring-cloud.

[29]https://github.com/Netflix/Prana.

[30]https://netflix.github.io.

[31]https://github.com/gliderlabs/registrator.

8 CONCLUSION

In this paper, we presented the AUTOGENIC approach to automatically generate self-configuring microservices. We introduced a novel approach to decouple software developers and operations personnel by separating their concerns. This leads to microservice development independent of the target runtime environment and thus also enables flexibility for operations personnel with respect to technological decisions and changes. Furthermore, we presented the AUTOGENIC method, which describes the steps to generate self-configuring microservices. The method is described on a conceptual level and thus applicable to any modeling language and runtime environment fulfilling the proposed assumptions. We validated our approach by implementing a prototype based on the TOSCA standard and state of the art technologies.

In the future, we plan to investigate use cases beyond dynamic configuration. Our prototype provides evidence that also monitoring features might be automatically enabled for developer-supplied microservices. Following the AUTOGENIC approach, required monitoring endpoints can be added in a transparent manner thus hiding the monitoring solution employed from software developers.

ACKNOWLEDGEMENTS

This research was partially funded by the Ministry of Science of Baden-Württemberg, Germany, for the Doctoral Program 'Services Computing'.

REFERENCES

Balalaie, A., Heydarnoori, A., and Jamshidi, P. (2016). Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, 33(3):42–52.

Fowler, M. (2017). Microservices Resource Guide.

Hüttermann, M. (2012). *DevOps for Developers*. Apress.

Kehrer, S. and Blochinger, W. (2018). Tosca-based container orchestration on mesos. *Computer Science - Research and Development*, 33(3):305–316.

Kookarinrat, P. and Temtanapat, Y. (2016). Design and implementation of a decentralized message bus for microservices. In *2016 13th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, pages 1–6.

Kratzke, N. and Quint, P.-C. (2017). Understanding cloud-native applications after 10 years of cloud computing - a systematic mapping study. *Journal of Systems and Software*, 126:1 – 16.

Lewis, J. and Fowler, M. (2014). Microservices a definition of this new architectural term.

Leymann, F., Breitenbücher, U., Wagner, S., and Wettinger, J. (2017). Native cloud applications: Why monolithic virtualization is not their foundation. In Helfert, M., Ferguson, D., Méndez Muñoz, V., and Cardoso, J., editors, *Cloud Computing and Services Science*, pages 16–40, Cham. Springer International Publishing.

Newman, S. (2015). *Building Microservices*. O'Reilly Media, Inc., 1st edition.

OASIS (2013). Topology and orchestration specification for cloud applications (tosca) version 1.0, committee specification 01. URL: http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs01/TOSCA-v1.0-cs01.html.

OASIS (2016). Tosca simple profile in yaml version 1.0, committee specification 01. URL: http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/cs01/TOSCA-Simple-Profile-YAML-v1.0-cs01.html.

O'Hanlon, C. (2006). A conversation with werner vogels. *Queue*, 4(4):14:14–14:22.

Pahl, C., Brogi, A., Soldani, J., and Jamshidi, P. (2017). Cloud container technologies: a state-of-the-art review. *IEEE Transactions on Cloud Computing*, PP(99):1–1.

Pahl, C. and Jamshidi, P. (2016). Microservices: A systematic mapping study. In *Proceedings of the 6th International Conference on Cloud Computing and Services Science - Volume 1 and 2*, CLOSER 2016, pages 137–146, Portugal. SCITEPRESS - Science and Technology Publications, Lda.

Schermann, G., Cito, J., and Leitner, P. (2016). All the services large and micro: Revisiting industrial practice in services computing. In Norta, A., Gaaloul, W., Gangadharan, G. R., and Dam, H. K., editors, *Service-Oriented Computing – ICSOC 2015 Workshops: WESOA, RMSOC, ISC, DISCO, WESE, BSCI, FOR-MOVES, Goa, India, November 16-19, 2015, Revised Selected Papers*, pages 36–47, Berlin, Heidelberg. Springer.

Stubbs, J., Moreira, W., and Dooley, R. (2015). Distributed systems of microservices using docker and serfnode. In *2015 7th International Workshop on Science Gateways*, pages 34–39.

Toffetti, G., Brunner, S., Blöchlinger, M., Dudouet, F., and Edmonds, A. (2015). An architecture for self-managing microservices. In *Proceedings of the 1st International Workshop on Automated Incident Management in Cloud*, AIMC '15, pages 19–24, New York, NY, USA. ACM.

Toffetti, G., Brunner, S., Blöchlinger, M., Spillner, J., and Bohnert, T. M. (2017). Self-managing cloud-native applications: Design, implementation, and experience. *Future Generation Computer Systems*, 72(Supplement C):165 – 179.

Turnbull, J. (2014). *The Docker Book: Containerization is the new virtualization*. James Turnbull.

Zimmermann, O. (2017). Microservices tenets. *Computer Science - Research and Development*, 32(3-4):301–310.