Co-Transformation to Cloud-Native Applications
Development Experiences and Experimental Evaluation

Josef Spillner¹, Yessica Bogado², Walter Benítez² and Fabio López-Pires²
¹Service Prototyping Lab, Zurich University of Applied Sciences, Winterthur, Switzerland
²Information and Communication Technology Center, Itaipu Technological Park, Hernandarias, Paraguay

Keywords: Cloud-Native Applications, Elasticity, Resilience, Continuous Development, Music Royalties.

Abstract: Modern software applications following cloud-native design principles and architecture guidelines have inherent advantages in fulfilling current user requirements when executed in complex scheduled environments. Engineers responsible for software applications therefore have an intrinsic interest to migrate to cloud-native architectures. Existing methodologies for transforming legacy applications do not yet consider migration from partly cloud-enabled and cloud-aware applications under continuous development. This work thus introduces a co-transformation methodology and validates it through the migration of a prototypical music identification and royalty collection application. Experimental results demonstrate that the proposed methodology is capable to effectively guide a transformation process, resulting in elastic and resilient cloud-native applications. Findings include the necessity to maintain application self-management even on modern cloud platforms.

1 INTRODUCTION

Users regularly expect software applications with outstanding features according to specific requirements, but also with indisputable technical qualities, including unlimited on-demand availability, constant high-quality behaviour independent from any interference or outside circumstances, as well as predictable costs which are aligned with the actual utilisation.

Modern cloud computing concepts and technologies facilitate the creation of such applications, but do not themselves guarantee them per se, and would thus not meet the requirements of SMEs and other users unless additional steps are taken (Wood and Buckley, 2015). Rather, application engineers are tasked to exploit these facilities to weave the desired qualities into the applications. This process is currently not dissimilar from a lottery. With a bit of luck, architects may create the right software architecture and engineers may correctly implement it. However, in order to make properly engineered cloud applications a commodity, a well-defined methodology to achieve Cloud-Native Applications (CNAs) is needed (Toffetti et al., 2017; Andrikopoulos, 2017; Yousif, 2017). Current research still focuses on transforming legacy version of existing applications but do not account for double effort when development and transformation to a cloud-native architecture occur in parallel.

In this context, the aim of this paper is thus to introduce a co-transformation methodology which starting from a sub-optimal and not fully reversible legacy application design through transformation during development achieves the desired characteristics in heterogeneous cloud computing environments.

To demonstrate the proposed methodology, we introduce HENDU¹ as running example. HENDU is an unfinished, continuously developed web-based application to record and analyse music played at arbitrary locations, events and radio stations (see Section 4). This considered application was designed and implemented taking into account some cloud principles and could be initially classified as a cloud-enabled application at the start of our work (see Section 2).

In order to provide essential characteristics for its massive deployment in production, HENDU must include cloud-native characteristics such as elasticity and resilience, which are achieved by co-transforming the application from a cloud-enabled to a cloud-native application, considering an application-specific co-transformation concept derived from the generic co-transformation methodology proposed in this work. In order to validate that the co-transformed application fulfils the requirements during development, several experiments are part of the concept.

¹Guarani word that means listen.
According to (Toffetti et al., 2017), cloud-native characteristics considered in this work can be defined as:

- **Elasticity:** CNAs supports adjusting their capacity by adding or removing resources to provide a required Quality of Service (QoS) in face of load variation avoiding over- and under-provisioning. CNAs should take full advantage of cloud environments being a measured service offering on-demand self-service and rapid elasticity.

- **Resilience:** CNAs anticipate failures and fluctuation in QoS for both cloud resources and third-party services needed to implement an application to remain available during outages. Resource pooling in clouds imply that unexpected fluctuations of the infrastructure performance need to be expected and accordingly managed.

The remainder of this paper is structured in the following way: First, a model of cloud computing application maturity levels is introduced, and the generic cloud-native co-transformation methodology is proposed. Subsequently, a detailed analysis of the HENDU application scenario is performed and a concrete fitting co-transformation concept is derived. The resulting implementation and an experimental evaluation thereof are then shown in the performed experiments. Finally, conclusions of this work are summarised.

### 2 CLOUD APPLICATION MATURITY LEVELS

Even though cloud computing is a well-established research area, the work on cloud applications is often sidelined and not rigorous (Yousif, 2017). The lack of a model to describe the cloudiness of an application is particularly evident and vague terms such as cloud-ready are often seen. Andrikopoulos defines an entire lifecycle for engineering cloud-based applications (Andrikopoulos, 2017) but it applies primarily to new software development without considering legacy code bases. The Open Data Center Alliance (ODCA) defines a four-level model (Ashtikar et al., 2014) but assumes isolated applications. Based on previous work (Toffetti et al., 2017), we therefore introduce a model which captures the cloudiness aspect as maturity levels even though application engineering not necessarily proceeds through all levels chronologically.

The presented model (see Figure 1) assumed a four-level maturity evolution for cloud applications. The first level (legacy) encompasses legacy applications which have not been designed for cloud environments. Often, this type of application even predate the corresponding platform technologies and require a manual installation and sizing. The second level (cloud-enabled) comprises of applications which already ship in a ready-to-deploy format such as a virtual machine or a container. Yet such applications work as isolated units outside of the platform’s feedback loop. This is the starting point of the considered HENDU application. When instead they integrate with surrounding cloud services such as service brokers, databases or health checks which are provisioned on demand outside of the application scope, the next level is achieved (cloud-aware). Finally, if applications fully exploit all facilities cloud environments offer to maximise availability, elasticity and resilience through self-management, they are considered to have reached the final level (cloud-native).

![Figure 1: A generic model for cloud application maturity levels and associated criteria, contrasting ODCA’s model.](image)

Each level can be subdivided to account for the amount of effort required by the application engineer or operator for maintaining the application offered as a service. For instance, the detection of platform services and the re-binding may happen automatically through discovery, or manually through configuration settings. Thus, the levels are not fully discrete but rather indicate the progress on a maturity spectrum.

Eventually, a CNA should be elastic and resilient through a certain degree of self-management. On an architectural level, the self-management can be completely self-contained within the application logic, or partially or wholly outsourced to an application management platform, e.g. a container platform in cloud environments. The platform must then expose the same characteristics, being elastic and resilient.
3 TRANSFORMATION METHODOLOGIES

The specialised literature on Cloud-Native Applications and suitable architectures reports on several migration and transformation strategies. In this section, we briefly present an overview about the field to justify the need for a proper co-transformation methodology, and then proceed to present ours as main contribution of this work.

3.1 Existing Methodologies

Similar to the perspective differences between cloud computing operations and cloud applications engineering, there is still a gap between cloud-native computing infrastructure and platforms, such as the ecosystem defined by the Cloud-Native Computing Foundation and other industrial approaches, and the applications side whose progress is driven by academic research, often through experience reports.

The previous work of one of the authors on transforming legacy software applications to cloud-native ones consisted of a technological fitting to distributed container and configuration management systems at that time (Toffetti et al., 2017). The application under transformation, Zurmo CRM, gained horizontal scaling and resilience capabilities which were validated experimentally over the period of almost two years.

Another experience report published in parallel by Balalaie et al. instead focused on reusability, decentralised data governance, automated deployment and built-in scalability, leaving little overlap in terms of defining the characteristics of the resulting cloud-native application (Balalaie et al., 2015). Additionally, a more recent architecture evolution progress proposal by Fowley et al. focuses on the feasibility of cloud-native transformations for SMEs (Fowley et al., 2017). It acknowledges the lack of in-house competencies and the need for re-engineering existing code due to the contained investment. According to the authors, the benefits will eventually outweigh associated costs.

A recent proposal, inspired by industrial research, is BlueShift which attempts to automate application transformation to cloud-native application architectures. The application functionality is discovered, analysed, transformed into artefacts and finally enabled as a cloud service (Vukovic et al., 2017). The method is leaning towards IBM’s BlueMix offering as target platform. It is not clear if this method, or any other, would work for generic targets.

In summary, existing methodologies to transform and lift applications to the cloud-native level do not yet consider migration for partly cloud-enabled and cloud-aware applications under continuous development. In consequence, this work introduces a co-transformation methodology, validating it through the migration of a prototypical music identification application, resulting in an elastic and resilient cloud-native application.

3.2 Co-Transformation Methodology

The proposed methodology consists of a number of transformation steps of which almost all can be executed in parallel to speed up the transformation in ongoing software development projects.

The necessary steps to achieve a cloud-aware application based on a cloud-enabled one are:

- The application must be aware about the cloud environment it is running in and about available associated capabilities for flexible use of cloud services and platform features to outsource critical secondary functionality (e.g. data storage). Standardised discovery and broker services such as the Open Service Broker API increasingly become enablers for the awareness.

- The application contains mechanisms for semi-static use of cloud management facilities, such as declaring rules for auto-scalers or load-balancers based on general metrics (e.g. CPU), as well as health checks and restarts.

To mature the application to a fully cloud-native level to maximise elasticity and resilience, the following additional steps are necessary:

- Conversion to a strictly separated set of stateless and stateful microservices as a set of types.

- Self-healing logic (in a basic form, restarts) for all microservices as a set of type instances in addition to platform features.

- Inclusion of sophisticated auto-scaling logic based on domain-specific metrics (e.g. response time) in addition to platform-defined metrics.

- Further self-management including policies to adaptively and autonomously enact the mechanisms for switching between application-controlled and platform-controlled services.

The methodology is not without alternatives. For example, it might be possible to create cloud-native applications without microservices. Given today’s trends in cloud platforms, such alternative designs would however be more costly and less native. Furthermore, the methodology does not mandate a location of implementation. Given the requirement of
adaptivity, the location may vary with the cloud environment. For instance, a cloud application hosted on Kubernetes may simply declare an auto-scaling rule which is processed outside of the application scope. The same application must however be prepared to execute in the same manner on other stacks and therefore needs portable platform services for the self-management which, even in a basic form, could serve as drop-in replacements for these mechanisms.

Summarising, Figure 2 shows the basic integration of an application under successive transformation to a cloud-aware and a cloud-native maturity level into a corresponding cloud computing environment. The cloud-nativeness is determined by the ability of the application to process environment information to make autonomous decisions about the degree of self-management.

4 HENDU CO-TRANSFORMATION

As previously described, the scenario application HENDU could be classified as a cloud-enabled application (see Figure 1) considering that its original version (cloud-enabled HENDU) is composed of a set of services, detailed in Section 4.1. These mentioned services are packed into Docker containers for high scalability (Dikaleh et al., 2016), being possible to run in platforms such as Google Container Engine, Amazon Elastic Container Service or even plain Docker computing environments.

Services in cloud-enabled HENDU still work as isolated units and required characteristics as elasticity and resilience are not achieved. Consequently, cloud-enabled HENDU must be transformed to a cloud-native HENDU to achieve current requirements for massive deployment. To demonstrate an application-specific application of the proposed methodology, Section 4.2 presents concepts and steps performed for the application co-transformation.

4.1 Original Cloud-enabled HENDU

Music royalties are often subject to fees on national composer or publisher rights organisations such as APA (Paraguay), SUISA (Switzerland) or GEMA (Germany). In this context, HENDU uses a music fingerprint database to heuristically determine the played songs, create a list of songs relevant to an event or radio station and finally submits it to the rights organisation for subsequent charging and billing.

HENDU (cloud-enabled) is composed of seven services: (1) Hash Generator, (2) Communication, (3) Radio Monitor, (4) Audio Recognition, (5) Filter, (6) Web User Interface and (7) Event Handler, as shown in Figure 3. Additionally it considers two database systems: (1) Operational (non-relational) and (2) Application Management (relational). The above mentioned components run as a single-instance container configuration without auto-scaling rules. As a main consequence, cloud-enabled HENDU does not achieve neither elasticity nor resilience.

Figure 3 shows the service architecture of cloud-enabled HENDU and its interaction with the external components for data acquisition as well as data visualisation for users. To be able to identify music, it is necessary to obtain the complete fingerprint of each music track. For that purpose, the Hash Generator service creates a set of fingerprints for each music $M_a$, dividing the original music $M_a$ in blocks of 30 seconds (i.e. $m_{f_a, i}$), considering an overlap of 15 seconds. The obtained set of fingerprints for each music track $M_a$ is stored in a Music Fingerprint collection and in a Musics table for further processing and comparison with collected samples. Music is collected for recognition from radio stations and events, as considered in the motivational examples of service operations presented as follows.

4.1.1 Music Recognition from Radios

The monitor of online radio stations is made through the Radio Monitor service that obtains audio samples and generates the fingerprint of each radio stream after a registration in Radios table for further processing and identification. The Radio Monitor service connects to the stream through the URL of radio $R_b$. Then, the mentioned service collects 30 second samples $s_j$ on MP3 format to generate a sample fingerprint $s_{f,j}$ for each collected sample $j$ of the radio $R_b$. The generated sample fingerprint $(s_{f,j})$, radio name $(R_b)$, radio station location and a time-stamp are stored in the Sample Fingerprint collection.

Once a fingerprint $s_{f,j}$ is generated and stored in the Sample Fingerprint collection, the Audio Recognition service compares it with existing music fingerprints (e.g. $m_{f,i}$), previously created with the Hash Generator service, resulting in a list of possible matches sorted by an affinity value called score. The
identified music with the highest score is stored in the Raw Results collection as a suitable match.

Based on the results of the Audio Recognition, the Filter service analyses and defines what music is played. The service creates threads for each radio, ensuring the concurrency for a faster process. Each thread $T_k$ processes the set of documents $d_{km}$ in the Raw Results collection that corresponds with its own assigned radio $R_k$. The thread $T_k$ reads the music of the first document $d_{k1}$, and the amount of blocks $q_m$ of that music, previously stored in Musics table and defined by its duration, then search’s for a match in the next $q_m - 1$ documents. If a match threshold is reached, the filter removes all the set $d_{k,q_m}$ of the collection, and considers that the music was played, then save this information on the Results table. But if not, the filter removes only the document $d_{k,1}$ and the cycle start again with the next document $d_{k,2}$.

4.1.2 Music Recognition from Events

At the same time that the system monitors online radio stations, it can obtain audio from events, having a similar process. The purpose in events is to obtain and process audio gathered from the environment through sensors $S$, for the creation of sample fingerprints $s_{fe,i}$. These are subsequent sent to the server for recognition. The sensor $S$ establishes a communication with the system through the Communication service using a Transmission Control Protocol (TCP) connection with an application-specific protocol, designed to handle multiple clients in a single instance. The connection between the sensor $S$ and the service is made through three steps. The sensor $S$ sends a request with its International Mobile Station Equipment Identity (IMEI) for habilitation. With IMEI the service consults Devices table and sends the response. Once the $S$ receives the response, it sends another request asking for a specific port. The server responds to this and opens a dedicated port to receive data from $S$. this mean that another sensor cannot connect through the same port assigned to $S$. The sensor sends the sample data and the service stores it in the Sample Fingerprint collection, closing the assigned port. The rest of the process is similar to online radios, with the difference that the Filter service creates threads and processes documents according to Devices table.

4.1.3 Data Visualisation

The main purpose of HENDU is to give information to musicians about music played in different events and online radio stations. Musicians can login to the system through the Web User Interface service and visualise which musics were played and where, as well as other relevant information registered. The Event Handler service establish a web socket connection, this is necessary for consults the Users tables for the login of the users and obtains the necessary information from the Results table.
4.2 Transformation Concept

Following the co-transformation methodology, a HENDU application-specific concept is derived. First, the application maturity needs to be lifted from cloud-enabled to cloud-aware, considering that:

- **Cloud-Aware 1 (CA1)**: Applications must be able to flexibly switch between self-managed and provider-managed services. As cloud awareness requires a conscious use of platform services, an explicit awareness about whether HENDU needs to launch its own data handling services, such as containerised databases, or bind to third-party services within or even beyond hosting platforms, such as Database as a Service (DBaaS), is needed along with a suitable reconfiguration mechanism.

- **Cloud-Aware 2 (CA2)**: Addition of auto-scaling rules for main services. This step is required to basically fulfil the *elasticity* characteristic expected for HENDU. Typically, this would be a combination of dynamic rules for auto-scalers and static rules for the initial scaling following the method of Ramírez López (López and Spillner, 2017).

With these steps, the application becomes cloud-aware as it integrates with stateful services of the environment and becomes itself entirely stateless. Subsequently, the maturity needs to be lifted further from cloud-aware to cloud-native, considering that:

- **Cloud-Native 1 (CN1)**: Services must be decomposed into qualified microservices (see Figure 4). To support the disposable nature of microservice instances, their implementations, for instance container images, should be as fast-booting as possible, which calls for more light-weight images.

- **Cloud-Native 2 (CN2)**: Provider-managed and self-managed health-checking for self-healing tools and restart mechanisms must be included for the application. This step is required to basically fulfil the *resilience* characteristic expected for HENDU.

- **Cloud-Native 3 (CN3)**: Addition of auto-scaling rules for all microservices based on domain-specific metrics. Auto-scaling rules based on general-purpose metrics (e.g. CPU utilisation bounds) are not applicable for cloud-native applications that must perform with an expected QoS defined particularly for the application.

- **Cloud-Native 4 (CN4)**: Provider-managed services could be limited for CNAs and consequently, CNAs should be able to independently and adaptively complement the cloud platform, and provide application-controlled services to avoid platform limitations (e.g. platform-controlled health checks).

5 IMPLEMENTATION

We have transformed HENDU into a cloud-native application following the proposed concept. Due to space limitations, this section reports on selected implementation aspects for CA1, CN1 and CN2.

5.1 Flexibility (CA1)

In today’s cloud platforms, the bindings to microservices are configured either by custom hostnames assuming authority over the Domain Name System (DNS) or by environment variables. DNS records can be updated any time during the execution but may be limited by incorrect Time-To-Live (TTL) settings, causing outdated records to be used.

Environment variables (e.g. MYDB=192.168.0.1) require a hierarchical launch of containers called master-slave model (Amaral et al., 2015) and furthermore require a restart of the container upon any change. In this context, Table 1 summarises the adaptivity options in two considered target cloud platforms.

Independent of the technical means to realise the bindings, there needs to be a systematic process to identify all binding locations, retrieve the list of candidate services, and perform the rebinding. For manual rebinding, this process needs to be made available to the developer through appropriate tooling. In our co-transformation work, we have designed a new tool to (i) parse HENDU’s Docker Compose files, (ii) identify service links within the composition, and (iii) offer to the developer to override the bindings. Thus, for any specified self-managed service, equivalent platform-hosted services may be used instead. Docker Compose would generate environment variables of the form `MYSQL=192.168.0.1`. Upon the developer decision to use a platform-hosted service instead, the self-managed service is removed from the composition and the equivalent environment variables are injected into all services previously depending on it. Additionally, to achieve CA2 we considered platform-managed auto-scalers based on CPU utilisation metrics as liveness probes for health checks in Kubernetes for service restarts.
5.2 Microservices Decomposition (CN₁)

We have realised CN₁ by proposing a new microservices architecture of cloud-native HENDU. In contrast with the original cloud-enabled HENDU architecture shown in Figure 3, the microservice architecture presented in Figure 4 considers several pods, composed by one or more containers for each microservice. These pods can be elastically replicated according to current demand. Each pod, except Filter, has a service layer to communicate with the others, represented in the architecture with dotted lines. Additionally, we have realised CN₁ by switching from Ubuntu Docker base images to Alpine images. Due to their small size, Alpine-based services are sometimes referred to as micro-containers. Across all image sizes, the reduction in size has on average been 81% (see Table 2) and consequently a reduction in taking snapshots and live migration time for containers has been significant as well. Measuring the time saved for these two management actions is out of the scope of this work and left as a future work.

5.3 Self-healing Method (CN₂)

Originally, self-management has been a crucial functionality within applications to survive faults and usage surges in the cloud. With the leverage of portable off-the-shelf microservice management frameworks, the alternative approach is to deploy those in conjunction with the application on plain disposable virtual machines or lower-level containers. Thus, the principle of dumb pipes, smart endpoints is evolved to dumb pipes, dumb infrastructure, smart endpoints which greatly simplifies the application design. Still, a bare minimum of self-management must be present within the application to account for failures within the platform. Consequently, the container composition of HENDU has been ported from Docker Compose to Kubernetes. In this context, Table 3 compares the self-management aspects of both platforms. What is apparent is the inability of both platforms to detect crashes and inconsistencies within the platform itself, leading to the need to include complementary application-level checks for high resilience. Over-
Table 2: Reductions in image size when considering microcontainers for microservices.

<table>
<thead>
<tr>
<th>HENDU Service</th>
<th>Ubuntu image size (MB)</th>
<th>Alpine image size (MB)</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communication</td>
<td>212</td>
<td>16</td>
<td>92%</td>
</tr>
<tr>
<td>Radio Monitor</td>
<td>771</td>
<td>166</td>
<td>78%</td>
</tr>
<tr>
<td>Hash Generator</td>
<td>770</td>
<td>165</td>
<td>78%</td>
</tr>
<tr>
<td>Audio Recognition</td>
<td>532</td>
<td>111</td>
<td>79%</td>
</tr>
<tr>
<td>Filter</td>
<td>544</td>
<td>118</td>
<td>78%</td>
</tr>
</tbody>
</table>

all, the co-transformation experience for HENDU is summarised in Tables 4 and 5 for the successively considered maturation levels (cloud-enabled to cloud-aware to cloud-native). The mentioned tables contrast the respective steps from the co-transformation methodology, from the derived application-specific transformation concept, and from the resulting considered implementation.

6 EXPERIMENTAL EVALUATION

We have installed HENDU in its various maturity levels for a side-by-side comparison into a container management platform hosted at the institutional data centre of one of the authors which provides cloud self-services to all research and teaching staff.

The system runs OpenStack and offers Ubuntu 16.04 VMs in a *m1.small* instance size configuration. A second control installation was set up at the federated cloud of the European Grid Initiative (EGI) for comparison and cross-checking. In both cases, the container composition was scheduled using Docker Compose and Kubernetes running atop the virtual machines.

Additionally, a Kubernetes cluster composed by 2 nodes with the following characteristics was installed:

- RAM: 3GB.
- CPU: 4 cores (AMD Opteron 23xx 2.4Ghz).
- OS: CentOS 7.

For monitoring and visualisation purposes the Heapster service with an Influxdb database as well as Kubernetes dashboard to retrieve metrics.

6.1 Platform-managed Elasticity (CA2)

To demonstrate that HENDU may elastically adjust its computational resources according to current demand, it was evaluated against a HENDU version without auto-scaling capabilities as the original cloud-enabled HENDU. To achieve this, a workload simulation was considered, taking into account JMeter as suitable benchmark tool. The application was configured to simulate a workload trace of 100 HTTP POST requests, sent using a uniform distribution in 5 seconds periods, over a duration of 10 minutes. Obtained results were measured considering the Average Response Time (AvRT) and the Number of Requests Correctly Sent per minute (RS/m).

Experimental results are summarised as follows:

- **HENDU with CPU auto-scaling:**
  - AvRT = 120 (ms).
  - RS/m = 2833.1.
- **HENDU without auto-scaling:**
  - AvRT = 2923 (ms).
  - RS/m = 1212.7.

As the obtained results reveal, even when considering a basic metric for auto-scaling such as CPU, applications can improve response times by adjusting computational resources according to current demand. Including more sophisticated domain-specific metrics that are particular for each different application may incur in better QoS. Achieving this type of auto-scaling is left as a future work.

6.2 Platform-managed Resilience (CN2)

Existing experimentation tools such as Chaosmonkey or MC-EMU allow for random termination of microservices of an application (Spillner, 2017). As our interest is finding out about the combined resilience of application and platform, we have designed and implemented an MC-EMU-inspired custom parameterised Docker container terminator which targets the process running in the container as well as the management processes of Docker.

Docker represents a container management platform with three kinds of operating system processes on the host system: the top-level process *dockerd*, as its child the instance handling process *containerd* and below it a per-instance process called *containerd-shim*. Through a restart policy, Docker is supposed to be resilient against crashes.

We have assessed the resilience of the Docker engine with a technique which forces a single or repeated termination of up to two of these processes. Specifically, we run the assessment in two modes: One
Table 3: Self-management features comparison.

<table>
<thead>
<tr>
<th>Self-Management Feature</th>
<th>Docker-Compose</th>
<th>Kubernetes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Termination detection</td>
<td>internal crash only</td>
<td>yes</td>
</tr>
<tr>
<td>Slowness detection</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Inconsistency detection</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Custom detection</td>
<td>yes (health check)</td>
<td>yes (liveness, readiness)</td>
</tr>
</tbody>
</table>

Table 4: Methodology, Concept and Implementation for Transforming *Cloud-Enabled* to *Cloud-Aware* Applications.

<table>
<thead>
<tr>
<th>Methodology (Sect. 3.2)</th>
<th>Concept (Sect. 4.2)</th>
<th>Implementation (Sect. 5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>The application must be aware about the cloud environment it is running in and about available associated capabilities for flexible use of cloud services and platform features to outsource critical secondary functionality.</td>
<td>Applications must be able to flexibly switch between self-managed and provider-managed services. As cloud awareness requires a conscious use of platform services, an explicit awareness about whether HENDU needs to launch its own services . . . or bind to third-party services . . . (CA1).</td>
<td>Manual implementation as a YAML configuration file in the deployment process. It contains references to endpoints and credentials which can be turned into links to platform-managed secrets for higher security.</td>
</tr>
<tr>
<td>The application contains mechanisms for selective static use of cloud management facilities, such as declaring rules for auto-scalers or load-balancers based on general metrics, as well as health checks and restarts.</td>
<td>Addition of auto-scaling rules for main services. This step is required to basically fulfil the <em>elasticity</em> characteristic expected for HENDU. Typically, this would be a combination of dynamic rules for auto-scalers and static rules for the initial scaling . . . (CA2).</td>
<td>Use of Kubernetes horizontal auto-scaling API with data provided by Heapster monitoring service, allowing auto-scaling based on Target CPU: 20%, Min Pods: 1 and Max Pods: 20.</td>
</tr>
</tbody>
</table>

Figure 5: Resilience of the Docker container platform – left graph: containerd-shim, right graph: containerd and containerd-shim.

which only targets *containerd-shim* and one which also targets its parent process with hard termination signals (SIGKILL) in successive order with breaks of 0.1s. After each termination sequence, the usability of the container under test is checked repeatedly, again using 0.1s intervals. For each number of termination attempts per sequence, 100 runs are performed. The results as shown in Figure 5 are clearly showing weaknesses within Docker which due to it being a core runtime also affects container management platforms such as Docker’s own Compose, Kubernetes and OpenShift. While few terminations do not show practical issues, repeatedly terminating *containerd-shim* eventually leads to exited containers with forgotten restarts (*missing* state). When also terminating *containerd*, containers rarely go missing but the Docker engine often ends up in a confused inconsistent state where the process list shows a container as running but Docker commands sent to it report about a non-existing container (*confused*). Eventually, in both cases the number of successful waits (*spin*), which use an exponential back-off strategy, is...
### Table 5: Methodology, Concept and Implementation for Transforming Cloud-Aware to Cloud-Native Applications.

<table>
<thead>
<tr>
<th>Methodology (Sect. 3.2)</th>
<th>Concept (Sect. 4.2)</th>
<th>Implementation (Sect. 5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conversion to a strictly separated set of stateless and stateful microservices as a set of types.</td>
<td>Services must be decomposed into microservices. To support the disposable nature of microservice instances, their implementations, for instance container images, should be as fast-booting as possible, which calls for more light-weight images (CN1).</td>
<td>Services were decomposed into microservices using Flask as a micro-framework for Python and changing the application specific protocols to REST for interoperability. Additionally, container images were also considerably reduced.</td>
</tr>
<tr>
<td>Self-healing logic (in a basic form, restarts) for all microservices as a set of type instances in addition to platform features.</td>
<td>Provider-managed and self-managed health-checking for self-healing tools and restart mechanisms must be included for the application. This step is required to basically fulfil the resilience characteristic expected for HENDU (CN2).</td>
<td>Use of liveness probe for the Kubernetes implementation and health-checks for the docker-compose implementation.</td>
</tr>
<tr>
<td>Inclusion of sophisticated auto-scaling logic based on domain-specific metrics … in addition to platform-defined metrics.</td>
<td>Addition of auto-scaling rules for all microservices based on domain-specific metrics. Auto-scaling rules based on general-purpose metrics … are not applicable for cloud-native applications that must perform with an expected QoS defined particularly for the application (CN3).</td>
<td>Left as a future work. For HENDU, application-specific metrics such as response time and instance count would be useful.</td>
</tr>
<tr>
<td>Further self-management including policies to adaptively and autonomously enact the mechanisms for switching between application-controlled and platform-controlled services.</td>
<td>Provider-managed services could be limited for CNAs and consequently, CNAs should be able to independently and adaptively complement the cloud platform, and provide application-controlled services to avoid platform limitations … (CN4).</td>
<td>Left as a future work. In conjunction with multi-tenancy and notifications from service brokers, cloud-native HENDU could optimise non-functional properties following complex requirements.</td>
</tr>
</tbody>
</table>

---

**Figure 6**: Improved resilience of the Docker container platform with revive container – left graph: 2s revive window, right graph: 5s revive window.

diminished significantly to around 0% and 60%, respectively.

#### 6.3 Self-managed Resilience (CN2)

In the next step, we have implemented and deployed an auxiliary (side car) container called Revive which
runs in privileged mode with control over the Docker command socket as part of the application. It autonomously monitors launched containers, learns about their names and launch commands, and replicates the exact setup in case the container goes amiss.

As there is a race condition between Docker’s restart attempts, the clean restart attempts at the end of each experiment rounds and Revive’s own attempts, the success of each attempt is carefully checked for and information about new instances is read from Docker’s process list in case of losing the restart race. With Revive, the number of missing containers when terminating only containerd-shim decreases significantly as shown in Figure 6. Furthermore, the average time for bringing the container back into service decreases significantly from 5.10s to 1.87s which greatly improves the overall application availability especially for delay-sensitive scenarios.

Figure 7 explains Docker’s container restart behaviour in greater detail. By default, a container is scheduled to be restarted after a cooldown period which is characterised by an exponential backoff. The period is reset after ten seconds of uninterrupted container runtime. A manual restart in the scheduled waiting period takes priority and leads to less cooldown, especially when greedily restarting containers already marked for restart. Some measurements did not terminate due to Docker health-checks enabled in these runs. When health checks are enabled, there is a likelihood that terminating the container from within does not succeed and the appropriate termination command hangs forever, followed by an inconsistent state in which a container is shown as running but cannot be referenced anymore. As result of the last experiment, Figure 8 shows that independently of the injected termination signal (SIGTERM or SIGKILL) within the container, there is a rate of around 34–37 inconsistent states after 1000 terminations, which is rather high (3.55%) and unacceptable for a production application. The results lead to a three-pronged recommendation for self-healing CNAs aiming at higher resilience on contemporary platforms.

1. Platform mechanisms to handle crashes of microservices should be complemented by application-controlled handlers for crashes and quality degradations in service implementations. These mechanisms are necessary but insufficient.

2. Due to immature microservice management platforms, consistency checks need to be performed and resolved, for instance by restarting the platform itself.

3. Portable microservice implementations should be implemented and deployed over multiple platforms to decrease the risk of failure, at the expense of additional network traffic and reduced volume discounts.

7 CONCLUSIONS AND FUTURE WORK

Software application engineering increasingly targets cloud computing environments for evident benefits in the quality level of the application delivery process. Our work, following the use case of the music royalty application HENDU operated in the cloud, has delivered a detailed analysis concerning systematic migration strategies through co-transformation and concerning resilient cloud-native applications in contemporary runtime environments. By systematically applying the proposed generic co-transformation methodology (see Tables 4 and 5), we introduced elasticity and resilience to the HENDU application.

Additionally, we have shown experimentally that outsourcing application-level self-management features, in particular self-healing to achieve resilience, to application management platforms is showing limited results. In particular, Docker, which also forms the
basis for Kubernetes and several other container platforms, does not protect against slowness of responses from microservices, and its protection against crashes and inconsistencies is limited due to missing self-resilience within the platform. Future work will include the consideration of additional fault types, improved automated derivation of scaling rules to cut down the engineering effort on their definition, and a strengthened co-transformation by considering development integration in modern cloud onboarding platforms such as the Kubernetes-based OpenShift.

Several future directions were also been identified to further advance this relevant research area. Our experiment code and raw data results are available for future research at https://osf.io/zsj7kj/.

REFERENCES


