World Wide Web (2019) 22:275–293
https://doi.org/10.1007/s11280-018-0562-5

SmartVM: a SLA-aware microservice deployment framework

Tianlei Zheng[1,3] · Xi Zheng[2] · Yuqun Zhang[1] · Yao Deng[2] · ErXi Dong[1] · Rui Zhang[3] · Xiao Liu[4]

Received: 5 February 2018 / Revised: 15 March 2018 / Accepted: 5 April 2018 /
Published online: 5 May 2018
© Springer Science+Business Media, LLC, part of Springer Nature 2018

Abstract Software-as-a-Service is becoming the prevalent way of software delivery. The popularisation of microservices architecture and containers has facilitated the efficient development of complex SaaS applications. Yet, for average SaaS vendors, there are a lot of challenges in managing microservices at a large scale while meeting the Quality-of-Service constraints. In this paper, we present SmartVM, a business Service-Level-Agreement (SLA) aware, microservice-centric deployment framework, designed to streamline the process of

✉ Xi Zheng
James.zheng@mq.edu.au

✉ Yuqun Zhang
zhangyq@sustc.edu.cn

Tianlei Zheng
tianleiz1@student.unimelb.edu.au

Yao Deng
dylan496517985@gmail.com

ErXi Dong
11749128@sustc.mail.edu.cn

Rui Zhang
rui.zhang@unimelb.edu.au

Xiao Liu
xiao.liu@deakin.edu.au

[1] Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China

[2] Department of Computing, Macquarie University, Sydney, Australia

[3] School of Computing and Information Systems, The University of Melbourne, Melbourne, Australia

[4] School of Information Technology, Deakin University, Geelong, Victoria, Australia

building and deploying dynamically-scalable microservices that can handle traffic spikes in a cost-efficient manner. We also compare our approach with traditional monolithic and the state-of-the-art microservice deployment approaches. The evaluation results show our approach advances in deployment cost, resource utilisation, and SLA compliance.

Keywords Microservice · Container · SLA · Autoscaling · Cost efficiency

1 Introduction

Software-as-a-Service (SaaS) applications have become increasingly popular for enterprises because they can alleviate the technical burden on organisations whose core business models do not involve IT [37]. Traditionally, software vendors deliver binary copies of their applications, and the consumers are responsible for running and maintaining the software, which implies the need for an investment in IT infrastructure and human resource [19].

Compared with the traditional, on-premises model, SaaS applications are offered with the hosting options included by the SaaS vendors. In many cases, the application components are not deployed onto the consumer's servers. This allows the consumers (tenants) of SaaS applications to focus on their business needs. On the other hand, with fully-controlled and less heterogeneous environments, the SaaS vendors can roll out software updates and security patches at a shorter interval [22]. Legal contracts between SaaS vendors and tenants, namely Service-Level-Agreements (SLAs) are often used to specify the expected Quality-of-Service (QoS) standards. The violation of SLAs can cause a penalty for the SaaS vendors [28].

Although from the tenants' perspectives, SaaS applications simplify technical issues in a cost-efficient manner, this does not remove the need for software maintenance. Instead, the responsibilities of software maintenance are handed over to the SaaS vendors. As a result, SaaS vendors often face numerous new challenges including security and privacy, scalability and resource optimisation, availability and fault tolerance, all rising from the fact that the SaaS vendors are responsible for hosting the applications for millions of end users (i.e., the users of the SaaS tenants). SaaS vendors are now becoming the middlemen between IaaS cloud providers (such as AWS, Microsoft Azure and Tencent Cloud) and the tenants, striving to find a balance point between minimising cost on the infrastructure and maintaining the required QoS. In this paper, we examine those challenges from the SaaS vendors' perspective, i.e. how to utilise cloud resource more efficiently while maintaining SLA compliance.

The popularisation of container deployment environments (e.g. Docker [7]) and microservices architecture have alleviated some of these problems. Microservices are small, independently running modules that are often deployed separately and developed by different teams within a large organisation. Microservices architecture mandates loosely-coupled components which run autonomously and communicate with each other using messages. There is no standard yet which defines what constitutes a microservice and the granuality of a microservice [8]. However, as we will show later in this paper, the granuality of each microservice can have a significant impact on the performance and cost efficiency in the deployment environment.

Microservices nowadays are often deployed to containers. Containers (with the Linux-based Docker being the most popular one) provide isolation to applications usually by using differential file system, Linux namespaces and cgroups. The key benefits of containers are their low overhead, the capability of running each container with an isolated filesystem and

allocated system resources, and on top of the same operating system. This greatly facilitates finer control over hardware resource allocation and improves the cost efficiency of large-scale microservice deployments.

Compared to monolithic deployment approaches, microservices architecture often leads to smaller, more maintainable software, and consequently smaller teams and more rapid development and deployment cycles [8]; on the other hand, containers have smaller resource overhead than virtual machines and can therefore scale in and out more responsively. Docker also makes it easier to have consistent development and deployment environments, which significantly reduces the cost of managing dependencies and gains popularity among software developers.

While containers allow SaaS vendors to deploy microservices at a fine granularity, the dynamic nature of microservices leads to new challenges given the scale and complexity of SaaS applications. Specifically, the major challenges include:

- **Uncertainty in load** The business models of most SaaS products imply that the SaaS vendors, who host the applications, are usually unaware of the business activities and lack the domain knowledge of host applications' specific business demands [34, 35]. For example, imagine a restaurant chain *R* using an online ordering system *S* developed and hosted by SaaS vendor *V*. When *R* launches a promotional campaign, it is unlikely and impractical for *R* to inform *V* in advance, so that *V* could prepare for the potential increase in traffic for *S*. It is also unrealistic for *V* to derive insights from *R*'s internal business information and models, such as customer statistics and seasonal income, in order to plan ahead. Instead, V needs to rely merely on the metrics of the application itself, not the business models and internal information of its tenants, to scale the service dynamically. Such limited information exacerbates the uncertainty in load.
- **Timeliness and accuracy of autoscaling** Conventional container orchestration platforms typically only offer limited rule-based autoscaling functionalities [1, 3, 20], where only resource metrics such as CPU usage are considered in those scaling mechanisms. While resource saturation possibly causes SLA violations, their exact correlation is yet to be explicitly discovered. An ideal autoscaling strategy is expected to react directly to application-level metric changes, such as increasing SLA violation rates. To the best of our knowledge, this is not currently possible without substantial modification of the infrastructure.
- **Separation of functional and operational concerns** Developers tend to be reluctant to evolve software systems. It possibly leads to a disruption in an organisation's operation if a deployment solution demands the refactoring of existing software systems. Therefore, it is crucial to automatically apply best practices in deployment and separate application developers' concerns from those of the operational engineers (who usually opt for the deployment changes).

Improper ways to handle these challenges can lead to under- or over-provisioning. If the resources provisioned are insufficient, the SLA requirements might be violated. On the other hand, allocating more resources than required incurs an unnecessary cost. It is difficult to precisely determine the appropriate resource demands for optimal provisioning, it becomes even harder when this has to be done continuously in real time.

Based on the observations above, in this paper, we propose an SLA-aware microservice deployment framework, namely SmartVM, to bridge the gap between the industry best practises and the developer-centric workflow of most SaaS vendors.

SmartVM tackles the issues above by splitting and reorganising microservices into a multi-tier architecture, where most resource intensive workloads (e.g. image rendering, log processing, etc.) are treated differently from the other user-facing and business-domain-bound operations (e.g. adding or removing orders, room booking, etc.). The traditional library functions and middlewares are turned into separate API Microservices that can scale independently. Business features are grouped into Business Microservices by their patterns of API access, resulting in less conflicting resource requirements.

Furthermore, SmartVM maintains SLA compliance by utilising application-level metrics for autoscaling. This allows SmartVM to more accurately and dynamically scale to accommodate fluctuating user traffic of SaaS applications while saving costs by reducing over-provisioning. To evaluate the proposed design, we implemented simulated business applications based on a real retail SaaS application. The business functionalities are implemented and deployed in three manners, namely the Uniform deployment, the Monolithic deployment, and SmartVM. The three deployments are then tested with different workloads, with their performance and cost efficiency being compared. The results indicate a significant performance advantage of SmartVM over the Uniform and Monolithic deployments, e.g., around 60% in cost reduction.

The rest of the paper is organised as follows: Section 2 introduces some of the related work; Section 3 presents the architecture and design decisions of SmartVM; Section 4 presents the results of our evaluation to validate the framework. Finally, Section 5 concludes the paper and points out the future work.

2 Related work

Autoscaling and load prediction In recent years, there has been a decent amount of studies on the elastic allocation of cloud resource to meet SLA requirements and to reduce cost. Queuing theory [14] is often used as a way to give formal proof or to create simulations for evaluation. The work in [27] uses machine learning to predict load and provide accurate, just-in-time allocation of resource, while incorporating factors such as the cost of reconfiguration. The authors of [33] uses fuzzy time-series and genetic algorithm to improve the accuracy of allocation even further. There are other methods for predicting time-series data in real-time through pattern matching such as [40]. The limitation of these studies is that they only consider hardware-level SLA instead of application-level requirements which are aligned with the SaaS tenants' business requirements and their customers' experiences. Moreover, they work on the granularity of machines or VMs, not on microservices and Docker containers.

Application-level metrics A few studies investigate on how to use application metrics to meet SLA requirements. The authors of [29] reports a dramatic decrease in SLA violation and resource efficiency as application-level metrics are incorporated into autoscaling algorithms. A low-overhead monitoring framework for application-level metrics is devised in [11]; and the approaches to implementing complex SLA monitoring logic using knowledge database are developed in [10]. However, these studies are tested in a VM-based deployment environment, and there is no discussion about the impact of splitting business features on SLA performance. Our prior work [25, 39, 44] have investigated various aspects of monitoring application-level metrics for microservice-based applications in theory. This work not only implements a SLA-aware monitoring and autoscaling middleware based on industry standard open-source tools but also evaluate the middleware against a real-world retail

SaaS application, the middleware and the evaluation test data are all publicly available at https://github.com/saasi.

Clustering of workload pattern The authors of [21] devise a formal framework to model microservices, features, and architecture. They develop MicADO, a genetic algorithm tool for automatically clustering microservices based on the access pattern derived from operational data. However, they only demonstrate reconfiguring microservice at a fixed scale. In addition, it still needs to be improved since in reality the features in microservices are not always separable, and reconfiguring the features within each microservices in runtime is likely to cause disruption to normal service. The work in [41] proposes a way to map application-level QoS requirements to the underlying hardware resource requirement so that the ideal execution parameters can be decided. These approaches are complementary to our SmartVM approach. But our focus in this paper is on the optimal way of autoscaling microservice instances in our suggested microservice architecture.

Auto-enforcement of best practices It is not effective for the solutions that aim at improving scalability and efficiency of the microservice deployment if a lot of efforts are required from the application-level developers to change their established programming paradigm. An industrial example, Netflix integrates best practices in the pipelines of their continuous delivery platform Spinnaker [13, 30]. In the open-source communities, OpenFaaS, a Function-as-a-service framework, alleviates developer burdens by building Docker images and manages autoscaling for them [24]. However, OpenFaaS is limited in scalability by providing a fixed-step rule-based autoscaling algorithm.

Compared with these relevant work, the contribution of our SmartVM architecture includes the following:

- It is specifically designed for Microservices and at a granularity of Docker containers.
- It is SLA-aware in monitoring and autoscaling
- Its multi-tier microservice architecture resolves resource conflicts and improves cost efficiency.

3 SmartVM architecture

SmartVM is proposed as an attempt to make an optimal solution in deploying SaaS applications. It can provide automation for deploying microservices in SaaS applications where development concerns can be separated from deployment concerns so that SaaS developers can compose Microservices to implement business functionalities without worrying about its runtime performance.

In this section, the first subsection (Section 3.1) provides an overview of SmartVM architecture. Section 3.2 illustrates the classification of workloads, and Section 3.3 elaborates the components of SmartVM.

3.1 Overview

The architecture of SmartVM is illustrated in Figure 1. We will first give a conceptual overview of the system, and the technical details of each component in Figure 1 are elaborated in Section 3.3.

As depicted in the diagram, applications are packaged as Docker container images and orchestrated by a standard container scheduling platform (in this case, Docker Swarm),

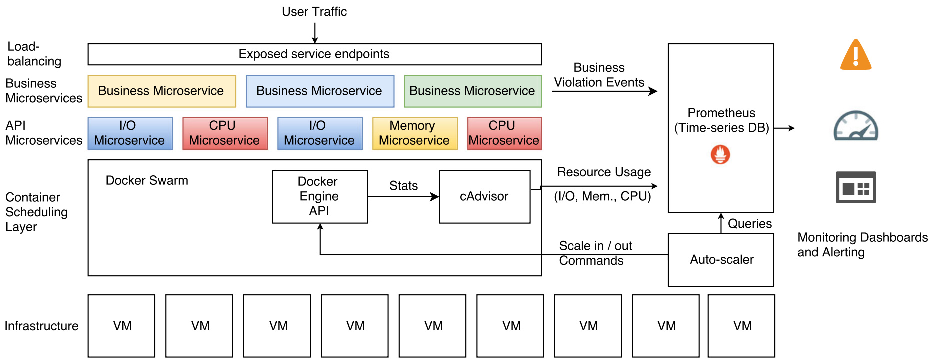


Figure 1 Overview of the architecture of SmartVM

which abstracts the underlying infrastructure such as virtual machines and cross-host networking. Each container can be considered conceptually as hosting an instance of a microservice. Note that even traditional monolithic applications can be containerised. Containers are fully compatible with other DevOps best practices such as Continuous Integration and allow SmartVM users to leverage existing, well-tested DevOps tools such as Jenkins [17] to continuously build, test and deploy applications, speeding up the development cycle.

The microservices in SmartVM can be conceptually divided into **Business Microservice(s) (BMS)**, which implements the business logic and are usually bound to certain business domains, and **API Microservice(s) (AMS)**, which implement resource-aware library functions. Correspondingly, the workload can be divided into BMS and AMS containers (see Section 3.2). In general, once a SaaS application workload is deployed, the **Autoscaler** monitors business SLA compliance for BMS and resource utilisation for AMS.

The user-facing functionalities are exposed by BMS through a gateway, which also acts as the first-tier load balancer. When requests come in, it is first handled by the load balancer, which then hands off the requests to one of the BMS containers. Whilst serving the user requests, BMS containers might need to make API calls to AMS containers in order to make certain low-level library calls. The communication between BMS and AMS containers is handled by a service mesh [36], which creates a virtual IP for each microservice and transparently routes requests to the relevant containers.

AMS containers run the tasks delegated by BMS containers and return the results to BMS containers. Because these AMS containers are usually resource-intensive, SmartVM collects metrics regarding CPU, memory and network usage of each container, and stores them in a time-series database. Subsequently, the Autoscaler queries the time-series database, and makes scaling decisions based on these metrics. Additionally, monitoring dashboards and alerting tools can also query the time-series database regularly, informing the administrators of the overall “health” of the system. Our monitoring algorithm is time-based and centralized but can be tailed to be event-based and distributed as in our prior work [43, 45].

Upon finishing the requests, BMS also checks whether non-functional business SLA requirements (such as time constraints) are met. If they are not, it logs a business violation event, which is then picked up by the log aggregator and enables the Autoscaler to decide whether more BMS containers are needed.

For both AMS and BMS, if the Autoscaler decides to scale out or scale in a microservice, it would call the scheduler, e.g., Docker Swarm via its API, to specify the required number of containers. And the scheduler would provision new containers for AMS or BMS. The **Load**

Balancer is also aware of the addition and removal of containers via Docker Swarm API, and the configuration of the **Load Balancer** is dynamically updated to reflect the changes.

3.2 The two-tier classification of workload

Organisational structure implications Conway's law [38] states that the software architecture is often correlated with the organisational structure of its developers. This correlation effect has been testified in some studies. For example, it is discovered in [23] that the software developed by loosely-organised open source communities has more loosely-coupled components than its commercial counterpart. The internal structure of most software organisations can be generally divided into product teams and research and development teams [16]. Generally, the product teams are responsible for developing software products with specific business features, which are usually closely related to customer experience, while the research and development teams provide generic software tools such as middleware, libraries and system-level services for the product teams. Accordingly, we propose a two-tier microservice structure in the SmartVM.

We separate the workloads on the SmartVM platform into two tiers (illustrated in the upper part of Figure 1):

– The first tier consists of **BMS**, which are often directly accessed by the users, and are generally not reusable outside of their business domains and do not require hardware specific resources directly.
– The second tier is made up by **AMS**, which can be further distinguished into CPU-, I/O- or memory-intensive microservices. AMS are usually not bound to business domains or directly accessible by the users but reusable across multiple applications. Some typical AMS include database access, PDF generation, image manipulation, training of machine-learning algorithms, and so on.

Figure 2 provides a logical overview of the two-tiered structure of microservices in SmartVM. Note that although not depicted in the figure, there are internal communications among microservices in the same tier. Such communication is less significant compared to inter-tier communication, which we try to optimise in SmartVM.

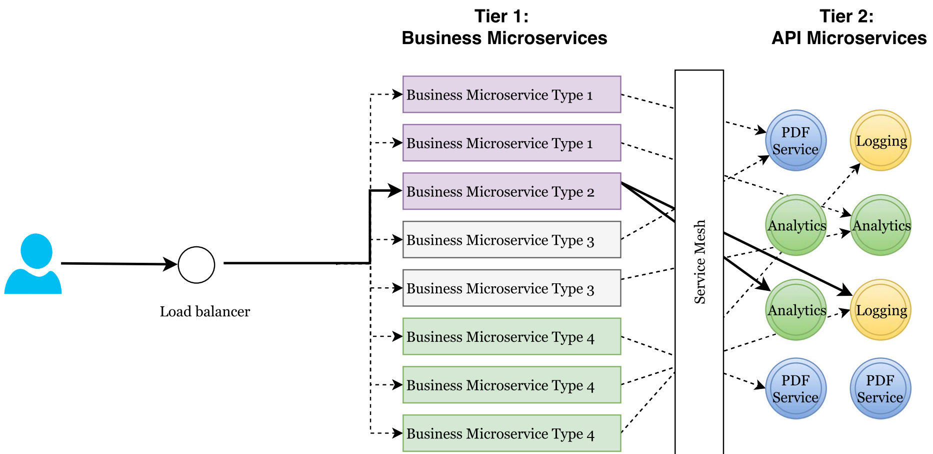


Figure 2 A logical view of the 2-tier microservice architecture in SmartVM

SmartVM is responsible for the monitoring, service discovery, networking, and autoscaling of all the workloads. Different microservices are associated with different scaling algorithms based on their characteristics. For example, for BMS, the compliance of non-functional business SLA requirements such as timing constraints, is taken into consideration by the **Autoscaler** (detailed below in Section 3.3).

3.3 Components

The SmartVM platform is built upon state-of-the-art SaaS deployment practices, most of which are strongly related to Docker containers with wide-spread supports of public cloud providers including AWS and Google. SmartVM enables multi-tier microservice deployments that are aware of business SLA requirements while being developer-friendly. The components involved in the architecture are listed below:

- **Scheduler**: As illustrated in Figure 1, underneath the microservices layers is the Scheduler that is responsible for scheduling and running containers on multiple physical nodes. At the current stage, we use Docker Swarm as the scheduler, which also provides a cluster-wide overlay network and built-in service discovery mechanism. The scheduler does not, however, make any decisions about whether, when and how to scale the microservice: it merely executes the scaling decision of the Autoscaler. Note that although some other components of SmartVM are drawn side-by-side to the Scheduler in Figure 1, they actually run as containers on top of the Scheduler.
- **Load Balancer**: The load-balancer is designed to equally spread the traffic across multiple instances of a specific microservice. Since we separate microservices in SmartVM into two tiers, different load balancing strategies can be applied to each tier. In this paper, we use Traefik [32], an HTTP reverse proxy and round-robin reverse proxy for the first tier (Figure 2) between end users and BMS. The second tier load-balancing, i.e. communication between BMS and AMS, utilises Docker Swarms' ingress network[36].
- **Instrumentation**: It is important to collect real-time information about the system for making auto-scaling decisions. In SmartVM, we collect both runtime numeric metrics and log messages of each container. Runtime metrics (stats) are retrieved from Docker Engine's APIs and pre-processed by cAdvisor [5].
    - **Numeric metrics** refer to those metrics of the application status that can be measured quantitatively. Some numeric metrics include the current number of active connections, total service requests, average response time, etc. For those metrics, we use Prometheus [26], a time-series database as the central storage, which pulls the metrics from the containers in the cluster and stores them for later querying. Each BMS or AMS, which handles the workload, can choose to expose numeric metrics for Prometheus to pull from, by using a client library and/or by a sidecar container, which runs side-by-side to the main container being monitored, and reads and transforms the application performance data to a Prometheus-compatible format [4].
    - **Log messages** are essential for accurate monitoring of compliance with business SLA requirements. Whenever a business violation occurs, an error message is written to `stderr` by the respective BMS. Such log messages are aggregated to a central monitoring location for the Autoscaler to make decisions. We use Fluentd [12] as the log aggregator and pre-processor, and Elaticsearch [9] for storage and log queries. Such settings allow us to easily

distinguish and keep track of log messages related to different business functionalities (as handled by BMS) and cope with the dynamic nature of SaaS applications.

- **Autoscaler**: The Autoscaler is a key component of the SmartVM architecture. It essentially involves a control loop which periodically checks the status of each microservice and makes autoscaling decisions based on several algorithms we developed. It is designed to be extensible such that different scaling algorithms can be included and integrated. It comes with a few following built-in autoscaling algorithms that application developer can choose to use by attaching labels to their containers to identify which algorithms suit their applications the best, either by the runtime or historical application status:
  - **state-based:** this autoscaling algorithm is inspired by Kubernetes' Horizontal Pod Autoscaler [20], the algorithm calculates the desired scale of a certain microservice, based on a given metric average value over a fixed duration of time. For example, in terms of CPU utilisation, the desired number of containers can be calculated as:

$$\text{Desired number of containers} = \frac{\text{Total average CPU usage}}{\text{CPU threshold for each container}} \tag{1}$$

    The advantage of the state-based algorithm is that it only takes one control cycle to reach the desired scale, hence making the system adaptive to burst surges and drops in traffic. The downside is that such algorithm is only suitable for metrics for which simple arithmetics (addition and division) operations are valid, and whose value is set to 0 by default when there is no load. The suitable metrics include CPU and I/O usage. However, such algorithm is not suitable for all scenarios.
  - **trigger-based:** this autoscaling algorithm checks whether certain predefined rules are met. When the metrics meet certain thresholds, e.g., the memory usage is above the limit, the business SLA violation rate is too high, a "scale out" rule is triggered, and a fixed number of new containers is provisioned for the microservice. "Scale in" works in a similar fashion. There is also a cool-down mechanism in place to avoid scaling too rapidly resulting in an unstable state.

    Trigger-based autoscaling algorithm can be applied in most scenarios. The downside is that with a burst surge in traffic, it might take multiple steps for a trigger-based algorithm to reach a suitable-sized number of containers. With machine learning algorithms, we can train the algorithm to adjust the "step size" based on the traffic pattern, so that scaling can be done with fewer steps, in order for a real-time performance to provision enough containers for the incoming traffic.

- **Visualisation and dashboards**: In order for system administrators and developers to watch the performance of the system in real time, as well as to inspect and localise bugs, a few dashboards are provided to show graphs on resource utilisation and business compliance. Specifically, Grafana [15] is used for displaying dashboards and Kibana [18] is used for inspecting logs. Note that such visualisation tools are optional: they do not store any information, nor do they play a functional role in the actual orchestration of microservices. Nevertheless, they are included here to demonstrate the best practice.

4 Evaluations

To evaluate the effectiveness of SmartVM architecture, we devise a series of experiments. A prototype is generated to simulate a generic real-world SaaS retail application for the evaluation purpose. In this section, the design and setting of the simulation experiments are depicted as well as discuss the outcome of the experiments.[1]

Evaluation application Our evaluation application is based on a flagship retail SaaS application from a public listed software powerhouse [31], which has a middleware and library team and a product team creating dedicated customer-facing business features. In our evaluation applications, the middleware and libraries are generalised as AMS and the business features are generalised as BMS. In order to ensure the workload generated from the evaluation application is as genuine as the real-world application, we simulate the following three types of functions to generate specific resource intensive workload respectively:

- *CPU-intensive workload* involves generation of random strings. We define one *unit* of CPU-intensive task to generate 10000 random strings that are 1000 characters long.
- *Memory-intensive workload* involves HTML parsing, that fetches the front page of http://news.ycombinator.com and parses the article list and returns an array of article titles and URLs. Specifically, One *unit* of memory-intensive task is equivalent to receiving and parsing the above-mentioned webpage once. A locally cached copy of the webpage is used as a fallback in case the network is unstable.
- *I/O-intensive workload* involves reading a configurable number of bytes (default is 1 MB) from a random file on disk and then returning it, causing both disk- and network-input/output streams. One *unit* of I/O-intensive task is equivalent to randomly reading and transmitting 1 MB of files.

Business operations 6 business operations are used in the simulation. They are combinations of the aforementioned task units with waiting (idle) time which simulates human-intervention (e.g. confirmation of orders) and calls to external services, as shown in Table 1. These simulated operations are modelled after a real retail SaaS application from the aforementioned SaaS provider [31].

Benchmarks We implement the above-simulated application features in three different architectures and deploy them in three different environments. The first and the second are used as benchmarks, while the third one represents the SmartVM architecture.

1. *Monolithic deployment:* As illustrated in Figure 3, in this benchmark architecture, all workload functions, together with business logic and the user-facing Web server are compiled into a single binary that runs as a single process. There is no network communication involved when the business logic utilises any of the workload functions. This simulates the architecture where all functionalities are compiled into a single, monolithic application. Intent-based autoscaling rules using the CPU and memory metrics are used for autoscaling this application.
2. *Uniform deployment:* As shown in Figure 4, The three business features are built into three different microservices that can scale independently. However, lower-level APIs

[1]Our solution prototype and corresponding evaluation dataset are publicly available at https://github.com/saasi/saasi-experiment and https://github.com/saasi/saasi-data.

Table 1 Descriptions of the simulated business operations

Operation	Description	SLA (Timeout)
#1	Wait 5 s.	10 s
#2	Parallelly perform 1 unit of CPU-intensive task and 1 unit of I/O-intensive task. Wait for both tasks to finish, then wait for another 2 s, prior to performing 2 units of memory-intensive tasks.	25 s
#3	Perform 1 unit of CPU-intensive task, wait for 5 s, prior to performing 1 unit of memory- and I/O-intensive tasks sequentially.	30 s
#4	Perform 1 unit of I/O intensive task, wait for 1 s, prior to performing 3 units of memory-intensive tasks.	20 s
#5	Perform 1 unit of memory- and CPU-intensive task sequentially	30 s
#6	Perform 1 unit of I/O intensive task, wait for 1 s, prior to performing 3 units of CPU-intensive task.	30 s

are still compiled into the microservices as libraries. Autoscaling is based on resource metrics, include CPU, memory and network throughput. Note that no monitoring of SLA-compliance is present. This simulates the state-of-the-art microservices architecture where applications run on top of an out-of-the-box container orchestration platform with no SLA-awareness.

3. *SmartVM:* Figure 5 shows how our approach differentiates from the uniform deployment, with the following key differences: 1) separation of business and API microservices, plus the splitting of API microservices' features based on access patterns; 2) monitoring of SLA compliance, where the autoscaling for Business microservice is based on application-level metrics, i.e. business SLA-violation rate, while the API microservices are monitored by the general, resource-based metrics.

4.1 Evaluation metrics

There are mainly 3 types of metrics that we collected during the evaluation:

1. *Resource utilisation:* We collect the basic resource utilisation metrics, including per-container CPU percentage and memory usage. These metrics reflect how "saturated" the hardware resource is in each running container. Within a reasonable limit, the higher values of these metrics indicate the better utilisation of provisioned hardware.

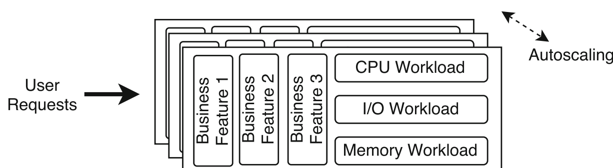


Figure 3 Monolithic deployment: business logic and low-level API libraries are compiled into a single binary

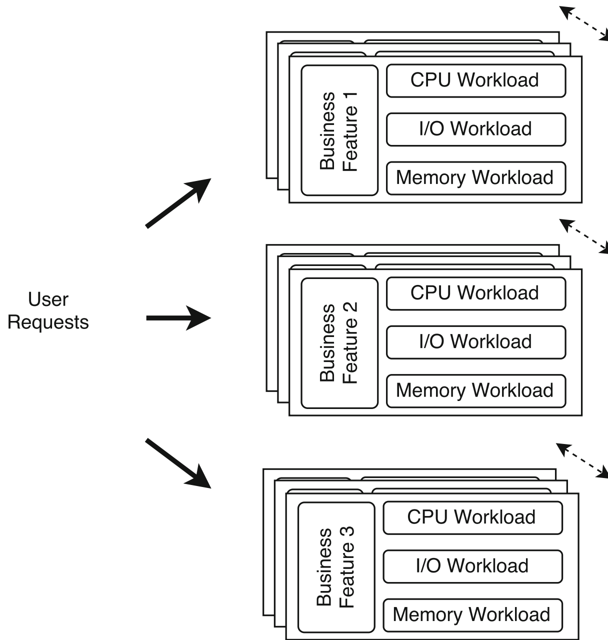


Figure 4 Uniform deployment: each distinct business logic and its used API libraries are composed into each independent running microservice

2. *Number of running containers:* On commercial cloud providers, such as AWS container service, the cost is usually associated the number of containers and how long each container has been running. Therefore, we use the number of running containers, averaged over time as a metric to depict how costly each deployment is.
3. *Occurrences of business violations:* We measure the average business violation rate, which is the number of SLA violation occurrences divided by the number of service requests from the applications in each deployment scenario. A lower value indicates better business SLA compliance.

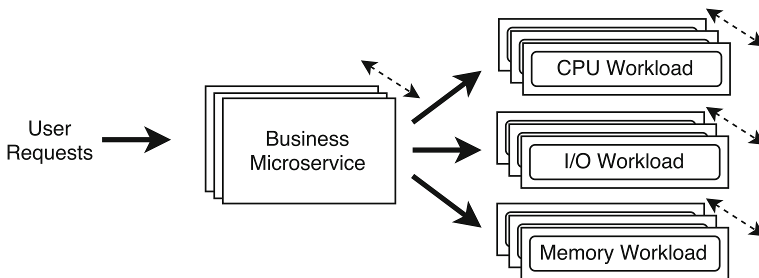


Figure 5 SmartVM: each distinct business logic operation is composed into a separate microserivce and API functions are split into 3 different kinds of microservices based on their resource utilisation pattern

4.2 Experiment setups

14 Virtual Machines on Tencent Cloud [6] are used for this experiment, with the following specifications:

- CPU: Intel Xeon E5-26xx v4 (2) @ 2.394 GHz
- RAM: 2 GB
- Hard drive: 64 GB HDD
- Networking: Tencent Cloud VLAN. 1000+ Mbps (ref)
- Operating System: CentOS Linux 7 (Core) x86_64
- Kernel: 3.10.0-693.11.6.el7.x86_64
- Docker Engine version: 17.09-ce

As for the containerised components, the software versions are:

- .NET Core: 2.1.4
- Elasticsearch: 5.6.6
- Fluentd: 0.12
- Traefik: Docker image `containous/traefik:b60edd9ee900`
- Prometheus 2.0.0

10 virtual machines are used for running the workload and 4 are used for running the supporting and management software.

4.3 Results

4.3.1 Deployment cost

All the three deployment approaches utilise containers under different resource configurations. In general, SmartVM can deploy microservices using containers with lower hardware specification, as a result of finer classification of microservices based on resource utilisation patterns. Specifically, in the experiment, we use 3 types of containers.

We are inspired by the pricing model of AWS Elastic Container Engine [2] to form a relative realistic pricing model for this evaluation. Since we are only interested in the *relative* costs of the three evaluation scenarios, we assume that for every unit of time, 1 vCPU is worth 16 units of cost, and 1 GB of memory costs 4 units. We assume a pay-as-you-cost model where the total cost is calculated as:

$$\text{Total cost} = \sum_{i=1}^{n} C_i \times T_i \tag{2}$$

where n is the number of container instances that has been provisioned (including those that has been terminated, C_i is the unit cost of container i, and T_i is the units of time container i has been running for. As shown in Table 2, the "larger" each container type is and the longer each container instance runs, the higher the cost turns out to be.

Figure 6a shows the cost of each component under different load (ranging from 10 to 250 concurrent users). *SmartVM* performs consistently better than the *Uniform* deployment, costing as low as 1/3 of the latter, which can be attributed to the fact that SmartVM resolves conflicting resource requirements (e.g. when a microservice consumes a lot of CPU resources but a very limited memory) by splitting them into two tiers and clustering business features, which improves cost efficiency. Although in some test cases, the *Monolithic*

Table 2 Simulation container types and unit costs

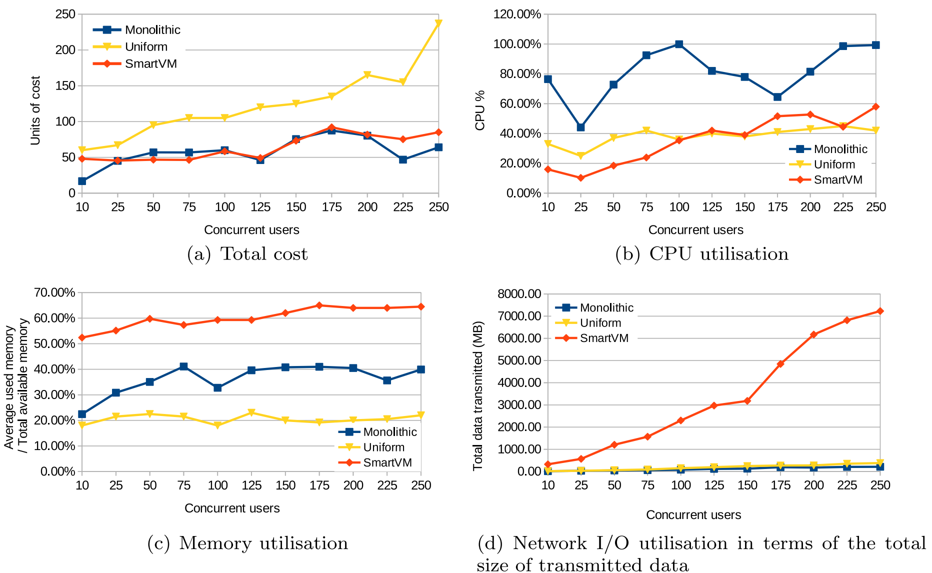
Container type	CPU	Memory	Cost (per unit of time)	Used by
Standard	0.4 vCPU	512MB	16.8 units	All applications / microservices in the *Monolithic* and the *Uniform* applications
Small	0.2 vCPU	256MB	8.4 units	Business Microservice in the *SmartVM* application
Small (low-memory)	0.2 vCPU	128MB	7.4 units	CPU Microservice in the *SmartVM* application
Small (low-CPU)	0.1 vCPU	256MB	5.2 units	I/O Microservice and Memory Microservice in the *SmartVM* application

deployment appears to cost slightly less than *SmartVM*, this comes at a price of much higher SLA violations (see Section 4.3.3).

In this experiment, we do not count Network I/O as a factor of cost, because most of the network I/O happens in the internal network (LAN) of the cloud provider, which is free of charge.

4.3.2 Resource utilisation

We measure resource utilisation rates by the weighted average of CPU and memory usage divided by the allocated quota for each container, where weights are derived based on the unit cost and the number of instances of each container type (as listed in Section 4.3.1). It is expected that for a costly container type, it tends to cause inefficient resource usage (i.e. not using it to the maximum).



(a) Total cost

(b) CPU utilisation

(c) Memory utilisation

(d) Network I/O utilisation in terms of the total size of transmitted data

Figure 6 Metrics of each deployment, in terms of the number of concurrent users

Figure 6b shows the CPU utilisation rates of each deployment approach. Despite being a little counter-intuitive, the *Monolithic* deployment has the highest utilisation rate of CPU resource. However, as in Section 4.3.3, the high SLA violation rate of the *Monolithic* application indicates that it is hard to justify its cost efficiency since in reality, the penalties for SLA violations surely surpass the savings on CPU resource. On the other side, The *SmartVM* application is less cost-efficient than the *Uniform* application under low traffic, possibly due to the fact that there are more microservices in the *SmartVM* deployment, and therefore more containers are initially provisioned. As the load increases, *SmartVM* starts to outperform *Uniform* deployment.

As for memory utilisation, Figure 6c shows that *SmartVM* performs consistently better than the other two approaches. Overall, *SmartVM*'s memory usually is 1.5 times more efficient than the *Monolithic* deployment, and more than twice efficient then the *Uniform* deployment. Also as explained in Section 4.3.3, *SmartVM* has the least violation of SLA.

Figure 6d illustrates the total network traffic of each deployment approach under different loads. *SmartVM* consumes significantly more network traffic than the other two approaches. This could be interpreted for the following reasons: 1) A large portion of the network traffic occurs inside the cluster LAN, which does not incur charges in most cloud providers. 2) Network difference is predictable since monolithic and uniform deployments are likely to have less much network I/O than SmartVM, because most invocations of library functions are conducted by internal function calls, while in SmartVM, BMS hands off intensive computations to AMS through network communications. It is important, however, to be aware that breaking down applications into microservices of finer granularity, especially as in SmartVM, demands stable and performance cluster networking, that can be easily solved in most state-of-the-art IaaS cloud providers.

4.3.3 Business SLA compliance

Figure 7 shows the performance of each evaluation application in terms of minimising SLA violations. Overall, SLA violation rates increase as the number of concurrent users increases. The *Monolithic* deployment performs better than the *Uniform* deployment under low traffic (where concurrent users $\leq$100) but the *Uniform* deployment outperforms the monolithic deployment in high-traffic situation. This shows the benefits of the microservices architecture. Furthermore, *SmartVM* perform consistently better than the other two

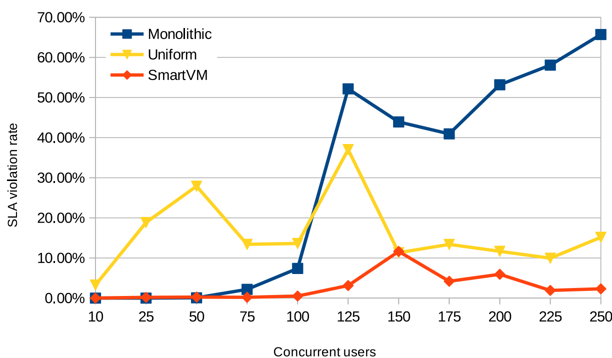


Figure 7 SLA violation rate versus the number of concurrent users

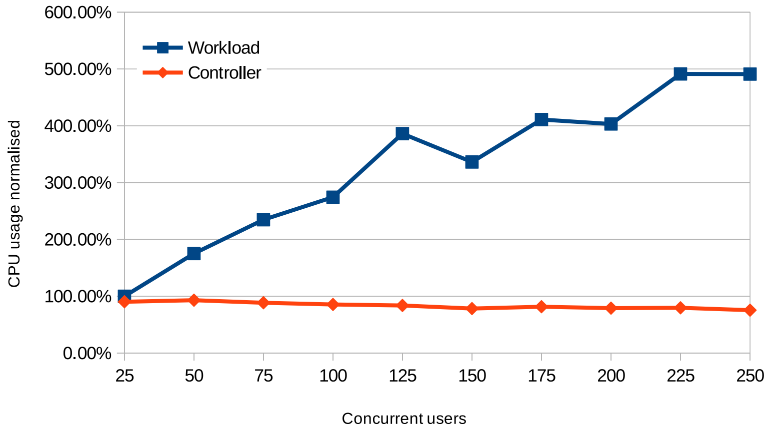


Figure 8 The growth of CPU usage of SmartVM components compared with the workload, as the load increases

deployment approaches, maintaining violation rates under 10% most of the time. This validates the effectiveness of application-level metrics-based monitoring in SmartVM, in that it can detect and react to SLA violations more promptly.

4.3.4 Overhead of SmartVM components

As in Figure 8, we also measure the overhead of SmartVM components. We normalise the CPU and memory usage of both the workload (the applications) and the "controllers" (SmartVM components) by dividing the usage metrics with their respective values in the lowest-load case (i.e. when the number of concurrent users = 25). When the traffic increases to 10 times of the initial load (i.e. when the number of concurrent users = 250), it is observed with a 391.0% surge in the CPU usage of the workload, but a mere 14.7% increase in the CPU usage of the "controllers". In terms of memory overhead, the figures (an 80.6% increase in workload versus a slight increase of 9.1% in the "controllers") similarly demonstrated that SmartVM does not incur an increasing overhead (rather, the resource consumed

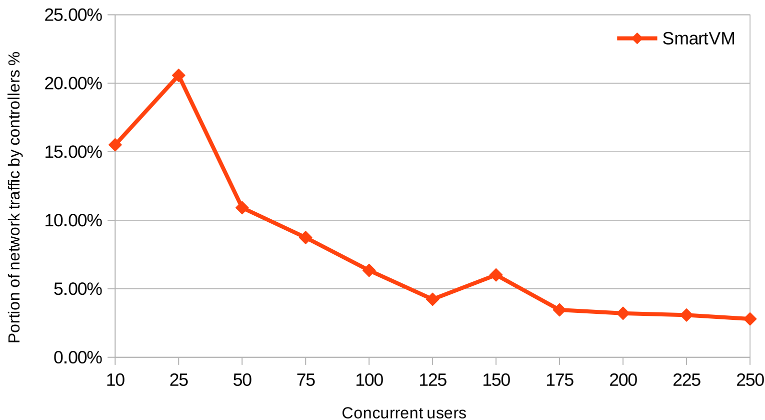


Figure 9 The network overhead of SmartVM components under different load conditions

by the “controllers” decreases relative to the workload) when the traffic increases, and therefore is fully scalable. Finally, Figure 9 shows that the overhead of SmartVM components drops below 4% as there are over 175 concurrent users.

5 Conclusions and future work

In this paper, we identify a number of challenges facing SaaS development and deployment, with a specific focus on SLA compliance. We proposed SmartVM, an SLA-aware, multi-tier microservices deployment framework, to ease the deployment workflow of microservices based on industry widely used open source repositories (e.g., Docker Swarm). The evaluation results indicate the advantages of SmartVM’s features, that are splitting microservices into multiple-tiers and further separating business microservices. SmartVM’s autoscaling mechanism and application-level metrics monitoring also demonstrate that minimising costs is achievable with low SLA violation rates. Our experiments show up to 66% cost reduction compared with the state-of-the-art uniform microservices approach with reduced SLA violation.

In the future, similar to our piror work [42], we will look into using machine learning techniques to automatically cluster business features and resource microservices based on their access patterns. This will serve as the next step to further alleviating the infrastructure maintenance overhead for SaaS vendors.

Acknowledgements This work was supported by the Ministry of Science and Technology of China (Grant No.2017YFC0804002). We thank Shengduo Chen, Changlong Wan and Shiwei Yan from the department of computer science and engineering, Southern University of Science and Technology, Shenzhen, China for their assistance to our work. Rui Zhang is supported by Australian Research Council (ARC) Future Fellowships Project FT120100832.

References

1. AWS Auto Scaling documentation (Accessed 10 Jan 2018). https://aws.amazon.com/documentation/autoscaling/
2. AWS Fargate Pricing - run containers without having to manage servers or clusters (Accessed 10 Jan 2018). https://aws.amazon.com/fargate/pricing/
3. Azure Autoscale (Accessed 10 Jan 2018). https://azure.microsoft.com/en-us/features/autoscale/
4. Burns, B.: The distributed system toolkit: Patterns for composite containers (Accessed 10 Jan 2018). http://blog.kubernetes.io/2015/06/the-distributed-system-toolkit-patterns.html (2015)
5. cadvisor. https://github.com/google/cadvisor
6. Cloud Virtual Machine - Tencent Cloud (Accessed 10 Jan 2018). https://cloud.tencent.com/product/cvm?language=en/
7. Docker. https://www.docker.com
8. Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R., Safina, L.: Microservices: yesterday, today, and tomorrow. arXiv preprint arXiv:1606.04036 (2016)
9. Elasticsearch. https://github.com/elastic/elasticsearch
10. Emeakaroha, V.C., Calheiros, R.N., Netto, M.A., Brandic, I., De Rose, C.A.: Desvi: an architecture for detecting sla violations in cloud computing infrastructures. In: Proceedings of the 2nd International ICST conference on Cloud computing (CloudComp‘10) (2010)
11. Emeakaroha, V.C., Ferreto, T.C., Netto, M.A., Brandic, I., De Rose, C.A.: Casvid: application level monitoring for sla violation detection in clouds. In: 2012 IEEE 36th Annual Computer Software and Applications Conference (COMPSAC), pp. 499–508. IEEE (2012)

12. Fluentd: Open-Source Log Collector. https://github.com/fluent/fluentd
13. Global continuous delivery with Spinnaker. https://medium.com/netflix-techblog/global-continuous-delivery-with-spinnaker-2a6896c23ba7
14. Gnedenko, B., Kovalenko, I.: Introduction to Queuing Theory. Mathematical Modeling. Birkhaeuser, Boston (1989)
15. Grafana. https://github.com/grafana/grafana
16. Griss, M.L.: Software reuse architecture, process, and organization for business success. In: Proceedings of the Eighth Israeli Conference on Computer Systems and Software Engineering, 1997, pp. 86–89. IEEE (1997)
17. Jenkins. https://jenkins.io/
18. Kibana. https://github.com/elastic/kibana
19. Kim, W.: Cloud computing: today and tomorrow. J. Object Technol. **8**(1), 65–72 (2009)
20. Kubernetes - Horizontal Pod Autoscaler. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/
21. Klock, S., Van Der Werf, J.M.E., Guelen, J.P., Jansen, S.: Workload-based clustering of coherent feature sets in microservice architectures. In: 2017 IEEE International Conference on Software Architecture (ICSA), pp. 11–20. IEEE (2017)
22. Lindholm, K.R.: The user experience of software-as-a-service applications. March **2**, 2007–005 (2007)
23. MacCormack, A., Baldwin, C., Rusnak, J.: Exploring the duality between product and organizational architectures: A test of the mirroring hypothesis. Research Policy **41**(8), 1309–1324 (2012)
24. OpenFaaS. https://www.openfaas.com/
25. Pham, V.V.H., Liu, X., Zheng, X., Fu, M., Deshpande, S.V., Xia, W., Zhou, R., Abdelrazek, M.: Paas-black or white: an investigation into software development model for building retail industry saas. In: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), pp. 285–287. IEEE (2017)
26. Prometheus. https://prometheus.io/
27. Roy, N., Dubey, A., Gokhale, A.: Efficient autoscaling in the cloud using predictive models for workload forecasting. In: 2011 IEEE International Conference on Cloud Computing (CLOUD), pp. 500–507. IEEE (2011)
28. Serrano, D., Bouchenak, S., Kouki, Y., de Oliveira, F.A. Jr., Ledoux, T., Lejeune, J., Sopena, J., Arantes, L., Sens, P.: Sla guarantees for cloud services. Futur. Gener. Comput. Syst. **54**, 233–246 (2016)
29. Souza, A.A.D., Netto, M.A.: Using application data for sla-aware auto-scaling in cloud environments. In: 2015 IEEE 23rd International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), pp. 252–255. IEEE (2015)
30. Spinnaker. https://www.spinnaker.io/
31. Tian Cai Shang Long Software. http://www.tcsl.com.cn/
32. Traefik. https://traefik.io/
33. Tran, D., Tran, N., Nguyen, G., Nguyen, B.M.: A proactive cloud scaling model based on fuzzy time series and SLA awareness. Procedia Comput. Sci. **108**, 365–374 (2017)
34. Tsai, W.T., Zhong, P.: Multi-tenancy and sub-tenancy architecture in software-as-a-service (saas). In: 2014 IEEE 8th International Symposium on Service Oriented System Engineering (SOSE), pp. 128–139. IEEE (2014)
35. Tsai, W., Bai, X., Huang, Y.: Software-as-a-service (saas): perspectives and challenges. Sci. China Inf. Sci. **57**(5), 1–15 (2014)
36. Use swarm mode routing mesh. https://docs.docker.com/engine/swarm/ingress/
37. Wu, W.W., Lan, L.W., Lee, Y.T.: Exploring decisive factors affecting an organization's saas adoption: a case study. Int. J. Inf. Manag. **31**(6), 556–563 (2011)
38. Yourdon, E., Constantine, L.L.: Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design. Prentice-Hall, Englewood Cliffs (1979)
39. Yu, D., Jin, Y., Zhang, Y., Zheng, X.: A survey on security issues in services communication of microservices-enabled fog applications. Concurrency and Computation: Practice and Experience
40. Zhang, Z., Jiang, J., Liu, X., Lau, R., Wang, H., Zhang, R.: A real time hybrid pattern matching scheme for stock time series. In: Proceedings of the Twenty-First Australasian Conference on Database Technologies-Volume 104, pp. 161–170. Australian Computer Society, Inc (2010)
41. Zhang, L., Zhang, Y., Jamshidi, P., Xu, L., Pahl, C.: Service workload patterns for Qos-driven cloud resource management. J. Cloud Comput. **4**(1), 23 (2015)

42. Zhang, Y., Zhang, M., Zheng, X., Perry, D.E.: Service2vec: a vector representation for Web services. In: 2017 IEEE International Conference on Web Services (ICWS), pp. 890–893. IEEE (2017)
43. Zheng, X., Julien, C., Podorozhny, R., Cassez, F.: Braceassertion: runtime verification of cyber-physical systems. In: 2015 IEEE 12th International Conference on Mobile Ad Hoc and Sensor Systems (MASS), pp. 298–306. IEEE (2015)
44. Zheng, T., Zhang, Y., Zheng, X., Fu, M., Liu, X.: Bigvm: a multi-layer-microservice-based platform for deploying saas. In: 2017 Fifth International Conference on Advanced Cloud and Big Data (CBD), pp. 45–50. IEEE (2017)
45. Zheng, X., Julien, C., Podorozhny, R., Cassez, F., Rakotoarivelo, T.: Efficient and scalable runtime monitoring for cyber–physical system. IEEE Syst. J., 1–12 (2016)