Engineering Cloud-based Applications: Towards an Application Lifecycle

Vasilios Andrikopoulos

Johann Bernoulli Institute for Mathematics and Computer Science
University of Groningen, the Netherlands
v.andrikopoulos@rug.nl

Abstract. The adoption of cloud computing by organizations of all sizes and types in the recent years has created multiple opportunities and challenges for the development of software to be used in this environment. In this work-in-progress paper, the focus is on the latter part, providing a view on the main research challenges that are created for software engineering by cloud computing. These challenges stem from the inherent characteristics of the cloud computing paradigm, and require a multi-dimensional approach to address them. Towards this goal, a lifecycle for cloud-based applications is presented, as the foundation for further work in the area.

Keywords: cloud computing, software engineering, cloud-based applications, software lifecycle

1 Introduction

The adoption of cloud computing has increased dramatically since the introduction of the term only roughly ten years ago — despite the fact that the technologies underpinning the paradigm have been around for a while longer. It is not an exaggeration to claim that in one way or another cloud computing offerings and associated technologies are currently being used by the majority of software-intensive enterprises. A report of the Thoughtworks Technology Advisory Board back in May 2015[1], for example, claims that *"Organizations have accepted that "cloud" is the de-facto platform of the future, and the benefits and flexibility it brings have ushered in a renaissance in software architecture."* From the thousand professionals from across sectors participating to RightScale's annual survey in early 2017 [31], 95% are reporting that the organization they belong to is already using or experimenting with the use of cloud computing.

Under the umbrella of the same term, however, there are multiple service delivery and deployment models on offer, succinctly summarized by NIST's widely accepted definition of cloud computing [24]. The availability of these options,

[1] Thoughtworks Tech Radar, May 2015: https://assets.thoughtworks.com/assets/technology-radar-may-2015-en.pdf

in conjunction with the plethora of offerings by cloud providers like Amazon Web Services (AWS), Microsoft Azure (MSA), and Google Compute Platform (GCP), and software solutions for the deployment of private clouds such as the ones from VMWware and OpenStack, create both opportunities and challenges for software developers [4]. Even the process of selecting an appropriate provider to run software on is an open research subject, with many of the issues identified in [34] (e.g. lack of standardization in the QoS descriptions and lack of long term performance prediction) still valid today. As such, there are still many issues that need to be resolved with respect to how cloud computing is to be used for software development.

At the same time, in the recent years the discourse on the best practices and principles of software development, at least in the industry, has been affected significantly by the introduction of two movements that have a co-dependence relation with cloud computing. The first one is the use of DevOps technologies and processes in order to bridge the gap between development and operations of software [8] in order to streamline software delivery and maintenance. The adoption of Continuous Delivery/Integration (CD/CI) techniques with frameworks like Jenkins[2] used together with deployment automation tools like Chef[3] or Ansible[4] shortens the development cycle dramatically and produces synergy with agile-oriented software development practices. Allowing for the management of multiple software stacks running in partially isolated containers inside one operating system as made popular by Docker[5], is the logical extension of this approach: each architectural component is developed, deployed, managed, and updated in its own software stack, and therefore it can follow a life cycle that is loosely coupled with the overall system evolution. This principle is made even more prominent in the second of the movements relevant to the discussion, i.e. microservices [27]. While there is an ongoing discussion in the academic community related to the actual innovation of microservices in comparison to Software-Oriented Architecture, it is important to notice how the notion of microservices have integrated into practice the use of design patterns, that so far have been mostly adopted at a much lower level (e.g. the Gang of Four book). Entries on microservices in Martin Fowler's blog[6], a popular grey literature source for practitioners and researchers provides many instances of this phenomenon.

In summary, therefore, the virtualization of resources and their offering as services, in conjunction with the DevOps movement, the containerization of software stacks, and the use of microservices, have evolved the way that software is developed, deployed, and managed over time. The key message of this paper is that *engineering software, and in particular software architecture, should similarly evolve.* For the purposes of scoping, the discussion is focused on how

[2] Jenkinks: `https://jenkins.io/`
[3] Chef: `https://www.chef.io/`
[4] Ansible: `https://www.ansible.com/`
[5] Docker: `https://www.docker.com/`
[6] For example: Microservices, by James Lewis and Martin Fowler (March 2014): `https://martinfowler.com/articles/microservices.html`

software engineering can change to incorporate cloud-related concepts by means of introducing a *cloud-based application lifecycle*. In the absence of a widely accepted definition of what constitutes one, and following the definition of service-based applications discussed in [3], this paper uses a working definition of *Cloud-based applications (CBAs)* as *applications that rely on one or more cloud services in order to be able to deliver their functionality to their users.* CBAs therefore include both cloud-enabled through migration [2] and cloud-native applications [21].

The rest of this paper is structured as follows: Section 2 identifies and presents the most relevant challenges to cloud-based application engineering (definitely not an exhaustive list). Section 3 transforms these challenges into a set of requirements on lifecycle methodologies in this context. Consequently, Section 4 discusses a CBA lifecycle that aims to address these requirements as the basis for future research. Finally, Section 5 compares the proposed lifecycle with related approaches, and Section 6 concludes with a short summary and future work.

2 Major challenges

Following the NIST definition [24], cloud computing has the following essential characteristics: (i) *On-demand self-service:* appropriate interfaces are offered to consumers to access resources (computational, storage, network, etc.) in an automated manner. (ii) *Broad network access:* resources are accessed over the network by heterogeneous clients. (iii) *Resource pooling:* service providers are enforcing a multi-tenant model of sharing the offered resources. (iv) *Rapid elasticity:* the volume of accessed resources can be adjusted dynamically, by any quantity and at any time. (v) *Measured service:* a metering mechanism is used to ensure appropriate billing for the used resources in predefined periods of time.

The combinations of these characteristics has severe implications for the software that is being developed in this environment. In the following we identify four major challenges that arise due to these characteristics.

2.1 *aaS software model

The first major challenge stems from the fact that resources are offered in the *Everything as a Service (*aaS)* model, usually affiliated with the categorization of delivery models into Infrastructure (IaaS), Platform (PaaS), and Software as a Service (SaaS), also covered by the NIST definition. The *aaS model is a natural outcome of the first two characteristics (i.e. *on-demand self-service*, and *broad network access*) and in many cases manifests as sets of RESTful APIs that are exposing cloud resources through relatively simple CRUD operations. While there has been lots of work on the subject of engineering service-based applications in the last 15 or so years, see for example [3], the very nature of service orientation still poses particular difficulties when used as the model for accessing resources. These can be attributed to the following:

Information hiding behind interfaces: exposing only the amount of information that is absolutely necessary for clients to use a service is one of the fundamental premises of service orientation [12]. However, this means that software developers have to refer to documentation and help desks in order to understand the boundary conditions and assumptions of consuming each resource.

Lack of control and observability over resource implementation: while the on-demand self-service characteristic prescribes a degree of control over the consumed resources by removing the need for administration on the part of the provider, this control is in practice limited to the operations defined in the service API, that for all practical purposes act as black box endpoints.

Distributed and heterogeneous environment: distribution transparency [33] is an essential feature of offered services, creating an impression of homogeneity and opaqueness to software developers. Nevertheless, the operating environment is fundamentally distributed, irrespective of the type of software developed on it (distributed or not).

Evolution driven by 3rd parties: as with many other API publishers in the past, cloud providers reserve the right to change their supported APIs at any point in time — and they do so for various reasons. As such, therefore, the evolution of software developed on these solutions is at least partially driven by the cloud providers and beyond software developer control.

Lastly, it can also be argued that while it is indeed possible to build all kinds of systems on top of cloud resources, it is consistent with the model that it is offered to design and implement them as services themselves. Doing so, however, imposes its own challenges, as evidenced by the continuous research output of the SOA community in the last two decades. The most thorny issue to deal with is probably the design of the system as services itself; indicative of the complexity of this issue is the fact that service design is identified as a major research question in both the SOA research roadmap [30], and its revision ten years later [10]. Further work towards this direction is therefore required.

2.2 Multi-tenancy of resources

One of the most difficult challenges to address, especially for performance-sensitive systems, is that of the shared nature of cloud resources due to its *resource pooling* characteristic. In a sense it is exactly this characteristic which makes rapid elasticity possible, while allowing for resource prices to be offered at very low levels, as also discussed by the next challenge. In essence, multiple tenants sharing the same infrastructure enable economies of scale for service providers and allow for higher utilization on the provider side through smart scheduling of large volumes of work load.

This sharing of resources, however, leads at the same time to performance variability that is external to the application itself, and as such outside of the control of the system developer. The inherent variance of cloud offerings has been documented in a series of publications: in [22], for example, large deviations are reported for similar in specification offerings across different providers, while significant variance can be observed in the same provider and offering within the same day and week [32] (and even more so across different availability zones), or even over the period of a year for the same offering [17]. Benchmarking cloud applications is faced with multiple challenges, see for example [9], and [14], and is not readily available as a tool for software developers to incorporate in their toolset. Cloud monitoring [1] is therefore the most common way to check and potentially address detrimental performance variation of the consumed resources.

2.3 Utility computing

One of the main reasons for the wide adoption of cloud computing is the transfer of costs from the capital to operating expenses through its "pay as you go" model [6], enabled by its *measured service* characteristic. In this sense, cloud computing can be seen as an implementation of the utility computing vision [39]. Access to computational resources in this context is enabled in a utility-oriented model, and results in the illusion of virtually infinite resources being available — assuming of course a sufficiently large budget [6]. At the same time, the use of economies of scale on behalf of the service providers, and the environment of intense competition for a very lucrative market, result into continuously decreasing prices for the offered resources. This creates the dynamics of a "race to zero" phenomenon, especially in storage offerings[7]. Even if the provider prices are not lower in comparison with operating one's own data center as e.g. in the (already outdated) analysis of [37], there are boundary conditions that still make the use of cloud solutions favorable to the alternative [38]. The key is in the *rapid elasticity* characteristic which allows for quick scaling to cope with dynamic demand, resulting in compensation of potentially incurred losses throughout relatively stable demand periods by means of serving requests that would otherwise be over capacity and therefore resulting in loss of revenue.

Nevertheless, cheap is not the same as free of charge, and costs for successful cloud-based companies might run so high that result in their profit margin shrinking to the point of necessitating the migration to their own data centers instead, as documented by the case of Dropbox[8], a company that was famous for running all their infrastructure on Amazon Web Services until that point. Rightscale's 2017 State of the Cloud survey [31] reports two stark findings that are relevant to this discussion: first, mature adopters of the technology are more concerned with cost management in comparison to beginners to it; second, only a minority of companies actually take measures to minimize unnecessary costs (e.g. VMs unnecessarily being active). Some notion of costs control is therefore clearly necessary.

[7] See for example: `http://www.computerweekly.com/microscope/news/4500271376/Whatever-the-cost-may-be-Cloud-price-war-continues`

[8] See `https://www.wired.com/2016/03/epic-story-dropboxs-exodus-amazon-cloud-empire/`

2.4 Distributed topology

There is no escaping the fact that systems developed in the cloud environment are essentially distributed, and they need to be designed, implemented, and operated as such [11]. Distribution in this case is both spatial and logical, but distribution transparency [33] is partially violated when e.g. availability zones are used for the deployment of applications. On top of this, there are multiple offerings by service providers that can be used as alternatives to application components [4] taking advantage of the *on-demand self service* characteristic. For example, Database as a Services (DBaaS) offerings can replace completely the data layer of an application, providing native scaling mechanisms to cope with increasing demand. An illustration of the range of possibilities available to software architects is the case of Netflix, which combines AWS EC2, S3, EBS and other offerings to run in a cloud-only environment[9].

Adding to the size of the design space is the capability to use containers as the means for enabling portability of application components and work loads across cloud providers, essentially expanding on the characteristic of *resource pooling*. In conjunction with a cloud orchestration layer, containers allow for a series of benefits like reduced (infrastructure) complexity, automation of portability, better governance and security management, transparent geographical domain-aware distribution, and the ability to automate services that offer policy-based optimization and self-configuration [23]. As a result, there are many possible system configuration options that are optimal under different dimensions [4], e.g. cost versus performance, creating exceptional challenges to software architects in identifying the best solution for their needs.

3 Requirements on the Solution Space

From the discussion above it becomes quickly obvious that addressing these challenges is a multi-faceted undertaking, and that their nature requires them to be considered throughout the lifecycle of software systems operating in the Cloud. The following constraints are, as a result, imposed on possible solutions for engineering *cloud-based applications (CBAs)*:

1. Irrespective of the purpose and type of software under consideration, cloud-based application development should *understand and incorporate service-orientation concepts*. In practice, this means that resources are accessed through programmatic interfaces, which in turn favors the Infrastructure as a Code approach [16] that homogenizes the way that the software itself and its supporting infrastructure is managed. Across similar lines, cloud service composition [18], which deals with the selection and aggregation of cloud services in order to support software, needs to be considered on equal grounds with (software) service composition [30] which delivers functionality by combining independent services.

[9] Netflix Global Cloud Architecture: `https://www.slideshare.net/adrianco/netflix-global-cloud`, slide 26

2. System design should incorporate the notion of *dynamic topology*. Topology here refers to the software and infrastructure stack required to operate the software artifacts under consideration, including e.g. the middleware associated with them. The system topology is prone to change over time due to changes *a)* in the system architecture, and *b)* refactoring of the infrastructure that supports the system. This might also include incorporation of new services by the same cloud provider or migration to another provider and/or deployment model. In this sense, the resolution of system architecture into concrete deployment models should rely on the generation of viable topologies through e.g. graph transformations, as per [4], instead of explicit modeling of alternatives. This is a consequence of the very large amount of available alternatives during design when considering all the different configurations available for each service type.
3. *Self-* characteristics* (e.g. self-management, -adaptation, -healing, -configuration, etc.) are necessary to deal with the multi-tenancy induced performance variability and its impact to the QoS of cloud-based applications. The introduction of a MAPE-K (Monitor, Analyse, Plan, and Execute over a Knowledge base) feedback loop [20] is a necessary and very common solution at this level as the means to implement control [29], but the difficulty is in evaluating the impact of individual cloud services, e.g. a DBaaS solution, to overall performance. Furthermore, the connection between run-time observations and design-time predictions is not sufficiently covered by the state of the art [15], and further work is necessary towards this direction. End-to-end performance measurement is potentially more important — alternative viable topologies have to be evaluated after all against their actual effectiveness in generating revenue — and in case of software delivered as services relatively easy to implement.
4. An *awareness of consumed resources* on self-management level during both development and operation of the system is essential. Cost models that cover the various deployment models, e.g. an extension of the model for hybrid clouds discussed in [19], should be used for this purpose. However, such analysis cannot be only performed offline. Instead, design- and run-time cost analysis should complement each other [25], resulting in cost models that are dynamically updated by actual billing data received from the cloud provider.

In the following we introduce a lifecycle model for cloud-based applications that incorporates the constraints discussed above as the means for defining in the future a holistic framework for engineering cloud-based applications. For this purpose the lifecycle model of service-based applications as discussed in [3] is used as the inspiration for this work.

4 Cloud-based Applications Lifecyle

4.1 The Phases of the Lifecycle

Figure 1 illustrates the proposed lifecycle of CBAs. Before proceeding with explaining the stages of the lifecycle, it needs to be pointed out that for the

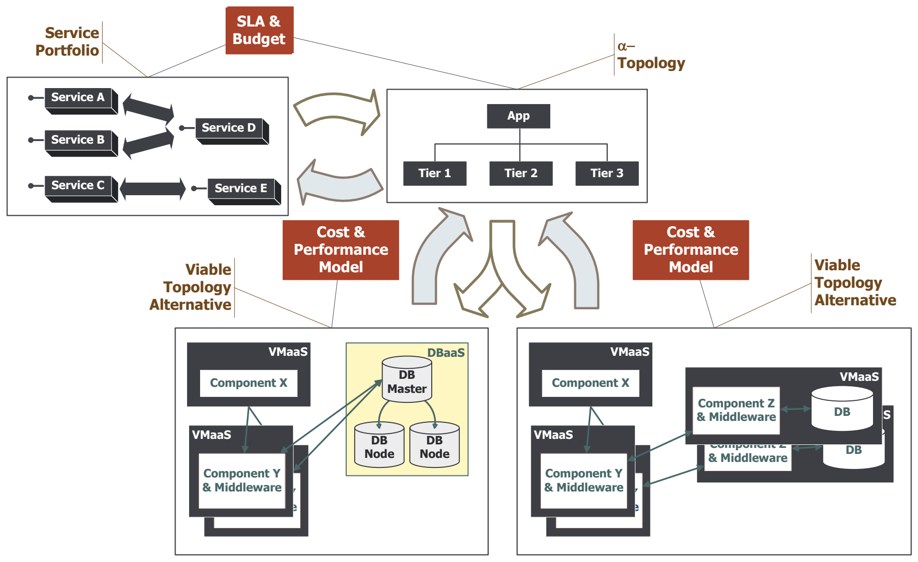

Fig. 1. The Lifecycle of Cloud-based Applications

purposes of this discussion, there is no clear design- and run-time (or development and operation, respectively) distinction, but more of a spectrum of activities spanning between them. The everything as a service and dynamic topology challenges affect more the one end (design), while performance variability and cost awareness more the other (run time). However it is impractical to attempt to assign them to specific stages of the lifecycle. The proposed CBA lifecycle (as shown in the figure) reflects this by intentionally not identifying when the transitions between stages are to take place, but only the transition relations between them. In this respect, the presented lifecycle is in accordance with the main principles of the DevOps movement [8] which unifies the different stages of software lifecycle.

Looking now at the figure, and starting from its top left part, the highest stage of the lifecycle consists of the *service portfolio* for the application, i.e. the collection of services that implement the functionalities offered by the application. Such services could be composed out of other services, belonging either to the same portfolio, or being external to it, as per the well established SOA practice [30]. Following the same principles, the service portfolio is the outcome of a *service identification* phase that connects higher level requirements and business operations into functionalities to be exposed by the application as services. In terms of how these services are mapped into software components and its supporting middleware, a *decomposition* into structural tiers can be applied using one of the methodologies discussed in [3]. In principle, non-application specific software components should be excluded from this process, resulting into a system architecture expressed as a set of *α-topologies* [4] (top center of Fig. 1).

The intentional exclusion of the underlying software stack from this stage (except where it cannot be avoided as e.g. in the case of customized middleware that needs to be rolled out together with the application) allows for flexibility in the transition to the next stage, that of *viable topology alternatives*, each one of which represent the whole software stack and its relation to the application components (bottom half of Fig. 1). Viable topology models encapsulate the various types of cloud services (e.g. VM or DB as a Service in the figure) that are part of the infrastructure supporting the software stack of the application.

As discussed above, and due to the numerous cloud service offerings currently available, a large number of viable topology alternatives potentially exist for each application. Selecting between them can be, and usually is interpreted as an optimization problem for which there are many techniques available (see [4] for further discussion). However, an alternative approach would be to look into this situation as an exploratory search problem instead. In this context, identifying a unique optimal solution in advance would be of not such interest as in transitioning between different alternative solutions in order to identify the optimal for the current conditions. For this purpose, the overall consumer utility and revenue generated by the viable topology currently used needs to be evaluated by comparing the continuously updated *cost and performance models* for each viable topology against the *Service Level Agreements (SLAs) and budget* associated with the service portfolio by the application owners. This approach requires, of course, that costs for the transition between viable topology models are negligible in comparison with the overall revenue generated by the application. Using a microservices-based approach for the decomposition of the service portfolio into isolated sub-systems before generating viable topologies would actually minimize such costs, since the finer granularity of each system tier would mean less components to consider (and potentially migrate) on the topology level. Alternatively, if this transition is deemed too costly and/or if the search space of viable topology alternatives has been exhausted then it is meaningful to revert to the previous stages of the lifecycle and either decompose the service portfolio as different α-topologies, or even refactor the service portfolio itself, repeating the cycle as necessary, as discussed in the following.

In order to add the necessary self-* mechanisms that regulate decision making during system operation a distributed MAPE-K model can be used [20], as discussed in Section 3. Considering the lifecycle of Fig. 1, however, it becomes clear that a hierarchical organization of controllers is better fitting. On the bottom level, it is possible to view the architectural components of each viable topology as its own autonomic element. However, all such elements need to coordinate with a controller on the level of the viable topology which is responsible for changes inside it. Another level of controllers is necessary to be added at the level of α-topologies when more than one viable topologies are active for a given decomposition. A similar process is repeated to the level of the service portfolio, and is used in order to trigger the transitions between stages of the lifecycle. Since the degree of automation that is feasible and available can vary among these transitions, it might become necessary to involve architects and system designers

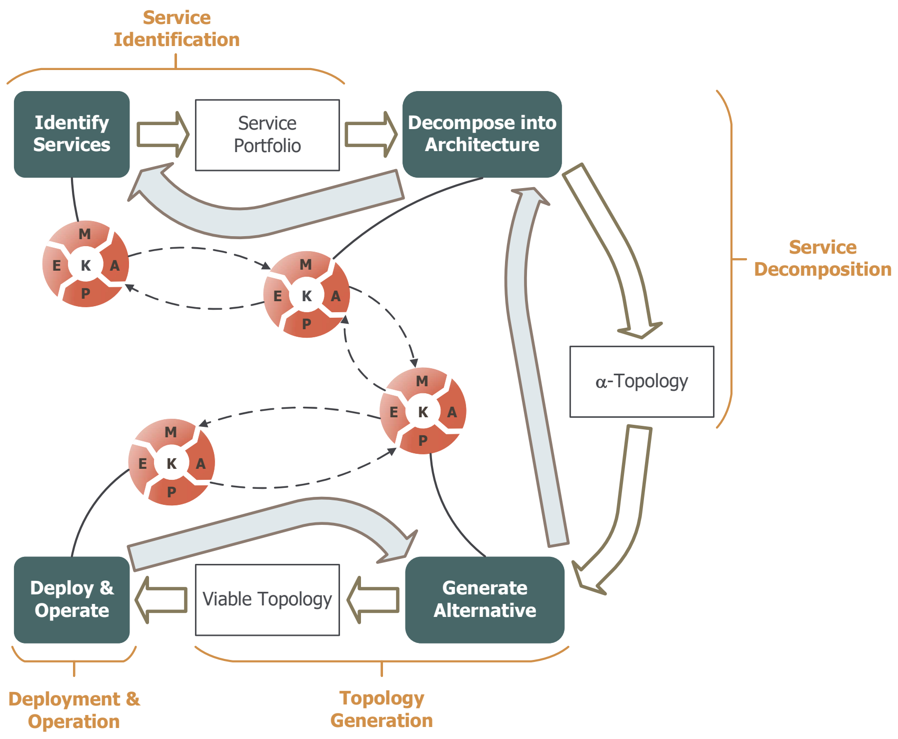

Fig. 2. The Phases of the CBA Lifecycle, with Activities Implemented as MAPE-K Loops and Information Flowing Between them

for this purpose. As such, design activities could be triggered by operations, as much as operational models could be derived during development.

Figure 2 summarizes and illustrates this discussion by identifying the concrete *phases* of the proposed lifecycle (Service Identification, Service Decomposition, Topology Generation, and Deployment & Operation) and the *activities* that take place in each phase (Identify Services, Decompose into Architecture, Generate Alternative, and Deploy & Operate, respectively). Each of the activities in the figure is implemented by a MAPE-K controller which is responsible for monitoring the situation at its level (e.g. α-topology), analyzing its behavior (is the application within its SLA and budget constraints?), planning for an action if necessary (deciding whether to transition into a new viable topology by moving into the Generate Alternative phase, or into a new *alpha*-topology by escalating the decision upwards into the Service Identification controller), and executing the decided action. Rules for the decision making, and the outcomes of past decisions are persisted in the knowledge base component of the controller at each level in order to learn over time about the effectiveness of each decision in a given context. Figure 2 shows the flow of information between the controllers of each level as

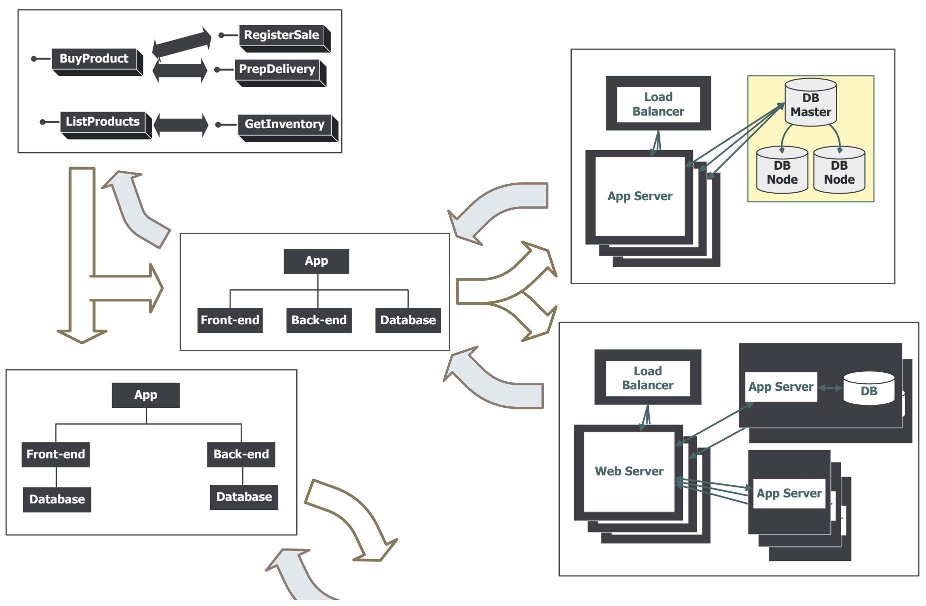

Fig. 3. The Lifecycle of an Example CBA (Web Shop)

dashed arrows between the loops. Bottom-level controllers (i.e. the controllers of the components in a viable topology in the Deployment & Operation phase) can only decide to escalate the need for an adaptive action upwards in the hierarchy, while top-level controllers (i.e. the controller at the level of Service Identification) can only trigger transitions into a lower level through the next phase.

4.2 An Example Instantiation

Figure 3 shows an example instantiation of the proposed lifecycle in the case of a Web Shop application. The service portfolio for the Web Shop consists (among others) of two client-facing services: BuyProduct and ListProducts, the former of which is composed out of services RegisterSale and PrepDelivery, while the latter one is using the internal service GetInventory. The BuyProduct service can be decomposed into a classic three-tier architecture, resulting in the top α-topology in the figure; for ListProducts a simpler two-tier architecture with separate (eventually synchronizing) databases is used. Staying with the first α-topology we can see that there are at least two alternative viable topologies to consider: in the first a DBaaS solution like AWS RDS[10] is used for the Database tier, operating in a cluster mode for scalability purposes. The front- and back-end are implemented as a web application deployed inside an *App Server* like JBoss[11]

[10] Amazon Relational Database Service: `https://aws.amazon.com/rds/`
[11] JBoss: `http://www.jboss.org/`

that is scaled horizontally by running inside multiple VMs in a service like AWS EC2[12]. A Load Balancer solution is deployed inside its own VM for traffic routing. An alternative viable topology for the Web Shop consists of deploying the front-end in its own dedicated VM cluster, decoupling the stateless functionalities of the back-end and deploying them separately in their own stack, and bundling the rest of the back-end into VMs combining application servers and database instances that replace the DBaaS solution (but which still need some logic to synchronize). Such transformations require of course much more detailed α-topologies than the examples in Fig. 3 that are kept to a minimum for illustration purposes, but are nevertheless possible to be largely automated given an appropriate knowledge base of reusable software stacks expressed e.g. as γ-topologies [4].

4.3 Evaluation & Discussion

Looking at the requirements identified in Section 3, it can be seen that the proposed lifecycle satisfies indeed them by: (i) seamlessly integrating service-orientation concepts both at the level of the artifacts that it deals with (applications as service portfolios), and at the level of cloud services used as the underlying resources for the deployment and operation of an application; (ii) building around the dynamic nature of application topologies by decoupling their α-topology from the actual viable topology and relying on the generation of the latter on demand to cope with changes in the perceived behavior of the application through the hierarchy of MAPE-K controllers; (iii) implementing the foreseen self-* characteristics by means of the same controllers; and finally, (iv) by introducing awareness of the consumed resources across the different phases of the lifecycle. However, validation of the lifecycle in more complex scenarios than the example presented in the previous through e.g. field studies, is the subject of future work since it is related with the development of the necessary tooling to support it (see Section 6). Furthermore, and in terms of limitations to the presented work there are two main issues not covered by the discussion: quality assurance for the developed software, and security and privacy. Both of these issues are in practice cross-cutting concerns running in parallel to the lifecycle, and while it can be argued that they could therefore be considered external to it, they nevertheless need to be examined further in future works.

5 Related Work

There are a number of mature works in the literature focusing on the complete lifecycle of cloud-based applications that are related to the lifecycle proposed here. In their majority however they address only parts of the requirements discussed in Section 3. For example, the Cloud Application Lifecycle Model (CALM) and its supporting framework is introduced in [35] without a provision for self-* characteristics or cost awareness. The same holds for [26] that discusses

[12] Amazon EC2: `https://aws.amazon.com/ec2/`

a cloud application lifecycle from a service governance perspective, and for the lifecycle presented in [28] which builds around the notion of blueprints as abstract templates for services to be published in application marketplaces. The work in [36] uses a centralized repository as the means to manage knowledge related to the phases of the lifecycle, but without the notion of cost awareness.

In further related work, the MODAClouds project relies on a Model-Driven Development-based approach to support the lifecycle of cloud-based applications [5]. The project builds on the models@runtime architectural pattern to connect run-time and design-time [13] and provides an IDE for the development, provisioning, deployment, and adaptation of CBAs. Nevertheless, the CBA lifecycle itself is only implicitly defined by this approach. The work in [7], part of the PaaSage project, discusses a service-based application lifecycle that emphasizes a multi-cloud deployment model. When compared to this work, the approach discussed in [7] attempts to (dynamically) optimize provider selection considering also monitoring data without however taking into account the possibility to re-distribute the application as part of this process.

6 Conclusions & Outlook

In summary, this work is based on the observation that the adoption of cloud computing, in conjunction with the advancements in software development in the form of DevOps, container-based software management, and microservices, requires an evoluationary step in software engineering practices, and especially in the area of software architecture. The challenges that drive this evolution are the everything as a service model in which cloud resources are offered, the multi-tenant environment created by resource pooling, the need to incorporate cost awareness due to the utility-based cost model for cloud computing, and the abundance of available offerings that can easily and efficiently replace parts of the software stack of each application. These challenges transform the lifecycle of cloud-based applications into a set of loops that transition between the set of application functionalities encapsulated as services, abstractly defined but application-specific architectural models, and software stack models that seamlessly incorporate cloud services. These transitions are triggered by controllers that coordinate within and across the various stages of the lifecycle.

Future work focuses on developing the methodologies and instrumentation necessary in order to support the proposed lifecycle, with a refinement of its various stages as an essential part of this process. A complete IDE in the manner discussed by the MODAClouds approach [5] is identified as the means to achieve this. Such an environment would further allow for field study-based validation of the lifecycle through collaboration with the industry. The development and integration with the IDE of the MAPE-K controllers as the implementation of the lifecycle phases-related activities is a critical component towards this effort.

References

1. Aceto, G., Botta, A., De Donato, W., Pescapè, A.: Cloud monitoring: A survey. Computer Networks 57(9), 2093–2115 (2013)
2. Andrikopoulos, V., Binz, T., Leymann, F., Strauch, S.: How to Adapt Applications for the Cloud Environment. Computing 95(6), 493–535 (2013)
3. Andrikopoulos, V., Bucchiarone, A., Di Nitto, E., Kazhamiakin, R., et al.: Service engineering, pp. 271–337. Springer (2010)
4. Andrikopoulos, V., Gómez Sáez, S., Leymann, F., Wettinger, J.: Optimal Distribution of Applications in the Cloud. In: Jarke, M., Mylopoulos, J., Quix, C. (eds.) Proceedings of the 26th Conference on Advanced Information Systems Engineering (CAiSE 2014). pp. 75–90. Lecture Notes in Computer Science (LNCS), Springer (2014)
5. Ardagna, D., Di Nitto, E., Casale, G., Petcu, D., et al.: Modaclouds: A model-driven approach for the design and execution of applications on multiple clouds. In: Proceedings of the 4th International Workshop on Modeling in Software Engineering. pp. 50–56. MiSE '12, IEEE Press, Piscataway, NJ, USA (2012)
6. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., et al.: Above the clouds: A berkeley view of cloud computing. Tech. Rep. UCB/EECS-2009-28, EECS Department, University of California, Berkeley (Feb 2009), http://www2.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html
7. Baryannis, G., Garefalakis, P., Kritikos, K., Magoutis, K., et al.: Lifecycle management of service-based applications on multi-clouds: a research roadmap. In: Proceedings of the 2013 international workshop on Multi-cloud applications and federated clouds. pp. 13–20. ACM (2013)
8. Bass, L., Weber, I., Zhu, L.: DevOps: A Software Architect's Perspective. Addison-Wesley Professional (2015)
9. Binnig, C., Kossmann, D., Kraska, T., Loesing, S.: How is the weather tomorrow?: towards a benchmark for the cloud. In: Proceedings of the Second International Workshop on Testing Database Systems. p. 9. ACM (2009)
10. Bouguettaya, A., Singh, M., Huhns, M., Sheng, Q.Z., et al.: A service computing manifesto: the next 10 years. Communications of the ACM 60(4), 64–72 (2017)
11. Cavage, M.: There's just no getting around it: you're building a distributed system. Queue 11(4), 30 (2013)
12. Erl, T.: SOA: principles of service design. Prentice Hall Press (2007)
13. Ferry, N., Solberg, A.: Models@ runtime for continuous design and deployment. In: Model-Driven Development and Operation of Multi-Cloud Applications, pp. 81–94. Springer (2017)
14. Folkerts, E., Alexandrov, A., Sachs, K., Iosup, A., et al.: Benchmarking in the cloud: What it should, can, and cannot be. In: Technology Conference on Performance Evaluation and Benchmarking. pp. 173–188. Springer (2012)
15. Heinrich, R., Schmieders, E., Jung, R., Rostami, K., et al.: Integrating run-time observations and design component models for cloud system analysis. In: Proceedings of the 9th Workshop on Models@run.time. vol. 1270, pp. 41–46. CEUR (2014)
16. Hüttermann, M.: Infrastructure as Code, pp. 135–156. Apress (2012)
17. Iosup, A., Yigitbasi, N., Epema, D.: On the Performance Variability of Production Cloud Services. In: Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on. pp. 104–113. IEEE (2011)
18. Jula, A., Sundararajan, E., Othman, Z.: Cloud computing service composition: A systematic literature review. Expert Systems with Applications 41(8), 3809–3824 (2014)

19. Kashef, M.M., Altmann, J.: A cost model for hybrid clouds. In: International Workshop on Grid Economics and Business Models. pp. 46–60. Springer (2011)
20. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer 36(1), 41–50 (2003)
21. Kratzke, N., Quint, P.C.: Understanding cloud-native applications after 10 years of cloud computing-a systematic mapping study. Journal of Systems and Software 126, 1–16 (2017)
22. Li, A., Yang, X., Kandula, S., Zhang, M.: CloudCmp: Comparing Public Cloud Providers. In: Proceedings of the 10th Annual Conference on Internet Measurement. pp. 1–14. IMC '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1879141.1879143
23. Linthicum, D.S.: Moving to autonomous and self-migrating containers for cloud applications. IEEE Cloud Computing 3(6), 6–9 (2016)
24. Mell, P., Grance, T., et al.: The NIST definition of cloud computing. NIST Special Publication 800-145 (2011), http://dx.doi.org/10.6028/NIST.SP.800-145
25. Moldovan, D., Truong, H.L., Dustdar, S.: Cost-aware scalability of applications in public clouds. In: Cloud Engineering (IC2E), 2016 IEEE International Conference on. pp. 79–88. IEEE (2016)
26. Munteanu, V.I., Fortis, T.F., Negru, V.: Service lifecycle in the cloud environment. In: Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2012 14th International Symposium on. pp. 457–464. IEEE (2012)
27. Newman, S.: Building microservices. O'Reilly Media, Inc. (2015)
28. Nguyen, D.K., Lelli, F., Papazoglou, M.P., Van Den Heuvel, W.J.: Blueprinting approach in support of cloud computing. Future Internet 4(1), 322–346 (2012)
29. Pahl, C., Jamshidi, P.: Software architecture for the cloud–a roadmap towards control-theoretic, model-based cloud architecture. In: European Conference on Software Architecture. pp. 212–220. Springer (2015)
30. Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F.: Service-oriented computing: State of the art and research challenges. Computer 40(11), 38–45 (2007)
31. RightScale: RightScale 2017 State of the Cloud Report (2017), https://www.rightscale.com/lp/state-of-the-cloud
32. Schad, J., Dittrich, J., Quiané-Ruiz, J.A.: Runtime measurements in the cloud: observing, analyzing, and reducing variance. Proceedings of the VLDB Endowment 3(1-2), 460–471 (2010)
33. van Steen, M., Tanenbaum, A.S.: A brief introduction to distributed systems. Computing 98(10), 967–1009 (2016)
34. Sun, L., Dong, H., Hussain, F.K., Hussain, O.K., Chang, E.: Cloud service selection: State-of-the-art and future research directions. Journal of Network and Computer Applications 45, 134–150 (2014)
35. Tang, K., Zhang, J.M., Feng, C.H.: Application centric lifecycle framework in cloud. In: e-Business Engineering (ICEBE), 2011 IEEE 8th International Conference on. pp. 329–334. IEEE (2011)
36. Tran, H.T., Feuerlicht, G.: Service repository for cloud service consumer life cycle management. In: European Conference on Service-Oriented and Cloud Computing. pp. 171–180. Springer (2015)
37. Walker, E.: The Real Cost of a CPU Hour. Computer 42(4), 35–41 (2009)
38. Weinman, J.: Cloudonomics: a rigorous approach to cloud benefit quantification. J. Software Technol 14(4), 10–18 (2011)
39. Zhang, Q., Cheng, L., Boutaba, R.: Cloud computing: state-of-the-art and research challenges. Journal of Internet Services and Applications 1(1), 7–18 (2010)