Experimenting with Liveness in Cloud Infrastructure Management

Pedro Lourenço[1,3], João Pedro Dias[1,2,3](✉), Ademar Aguiar[1,2,3], Hugo Sereno Ferreira[1,2,3], and André Restivo[1,2,3]

[1] DEI, Faculty of Engineering, University of Porto, Porto, Portugal
{pedro.lourenco,jpmdias,ademar.aguiar,hugosf,arestivo}@fe.up.pt
[2] INESC TEC, Porto, Portugal
[3] LIACC, Porto, Portugal

Abstract. Cloud computing has been playing a significant role in the provisioning of services over the Internet since its birth. However, developers still face several challenges limiting its full potential. The difficulties are mostly due to the large, ever-growing, and ever-changing catalog of services offered by cloud providers. As a consequence, developers must deal with different cloud services in their systems; each managed almost individually and continually growing in complexity. This heterogeneity may limit the view developers have over their system architectures and make the task of managing these resources more complex. This work explores the use of *liveness* as a way to shorten the feedback loop between developers and their systems in an interactive and immersive way, as they develop and integrate cloud-based systems. The designed approach allows real-time visualization of cloud infrastructures using a visual city metaphor. To assert the viability of this approach, the authors conceived a proof-of-concept and carried on experiments with developers to assess its feasibility.

Keywords: Cloud computing · Internet-of-things software engineering · Live programming

1 Introduction

The concept of cloud computing was predicted back in 1961 by John McCarthy. He stated that "computing may someday be organized as a public utility just as the telephone system is a public utility" [10]. However, it was only in the early 2000's that this prediction became a reality with the introduction of the Elastic Compute Cloud (EC2) developed by Amazon Web Services (AWS)—an Amazon subsidiary [3]—providing computing power in an on-demand self-service way. Currently, AWS offers more than ninety distinct services spread among twenty different categories [39], and more companies are providing this kind of services, *e.g.*, Google and Microsoft.

These services are typically made available to the general public, as what is known as public cloud hosting solution, in a pay-as-you-go fashion by the so-called Cloud Services Providers [7], who monitor, meter, and price the usage of resources, depending on service type and usage.

Different service models are offered by cloud providers, depending on the level of granularity and configuration that the developers require, being the following the most common [25]:

Infrastructure as a Service (IaaS). The cloud provider gives developers access to resources such as storage, networking, and servers in a pay-as-you-go fashion.

Platform as a Service (PaaS). Cloud providers offer developers access to a cloud-based environment on top of which they can build and deliver applications, abstracting the underlying infrastructure.

Software as a Service (SaaS). Service providers deliver software and applications through the Internet and users can subscribe to the software and access it remotely, *e.g.*, via a web portal or vendor APIs.

This paradigm reshaped how companies provide services, by allowing them to abstract, at different levels, from hardware infrastructure management, focusing on the virtual architecture and eradicating any possible concerns dealing with resource maintenance while improving manageability [7].

Alongside the reduced costs of using cloud computing when compared to on-premises solutions (*i.e.*, running on computers on the premises of the person or organization using the software), there is another key advantage: elasticity. The resources needed to meet the expected Quality-of-Service (QoS) can be rapidly provisioned, allowing the quick scale outwards and inwards to compensate for unpredictable business demands. This elasticity gives organizations more flexibility to focus on the core business instead of focusing on maintaining provisioned infrastructure. As a consequence, in most cases, there is no definite sense of location over the provided services beyond the ability, in some cases, to specify multiple higher regions which can be used as a strategy to increase reliability and avoid network outages [36].

As the market evolves, it becomes more demanding in terms of cloud services required, thus new "as-a-service" models start to emerge, leading to what has been called Everything as a Service (XaaS) [16,33] solutions (*e.g.*, Functions as a Service (FaaS), also known as Serverless [16,33,44]). As new paradigms emerge, in terms of connectivity and computation, they directly influence the market landscape with cloud providers offering even more services to fill the market needs. Internet-of-Things (IoT) as one of those paradigm-shifts, has lead diverse cloud providers to offer specialized services that answer the IoT scale, heterogeneity, and data-throughput needs. These services range from device management systems, to handle the ever-growing number of cloud-connected sensors and actuators, to specialized data analytics tools [11,42,44].

It is noticeable that cloud computing has brought several benefits for organizations in terms of interoperability and versatility, easing the process of meeting established QoS levels. However, there is a substantial amount of complexity in building and managing consistent and reliable infrastructures, resulting in the need of expert developers capable of implementing cloud-based systems [9].

In addition to the inherently complex nature of software systems [19,37], there is extra complexity in building systems within a cloud ecosystem. First, there is an ever-growing number of different services offered by cloud providers that make it harder to decide what is the best solution for a given problem [39]. Second, the final cost of a cloud solution can be highly volatile and hard to calculate *a priori* [14]. Third, there is no common taxonomy (or standards) among cloud providers, which leads to confusion and makes comparing solutions harder [14,55]. And last, the different services provided by each cloud vendor can lead to a vendor-lock that impacts an eventual process of migrating a solution, if needed [14,43,55].

As we move towards more complex cloud-based software systems, we will eventually come to an explosion of different services, each one managed individually, possibly leading to serious management challenges. This complexity makes it harder to understand cloud-based systems and the value that they bring to the business [31].

Even further, the 2017 edition of RightScale's State of the Cloud Report [46], an yearly survey on cloud computing trends, inquired 1002 IT professionals, and showed that when comparing the years 2016, 2017 and 2018, the most relevant cloud challenges are the lack of resources/expertise and security in cloud management (Fig. 1). Moreover, even though there is a decline in nearly all challenges compared with the previous year, it is interesting to note that governance/control is the only challenge that has almost stagnated in the three-year comparison.

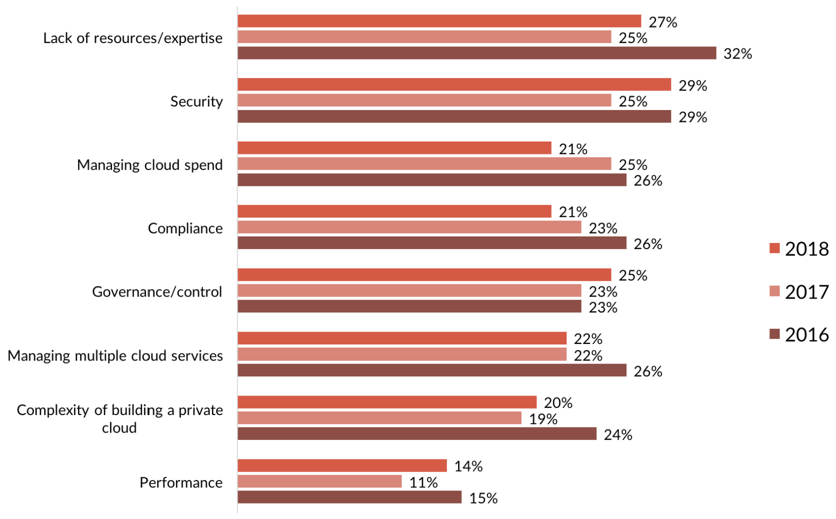


Fig. 1. Cloud challenges comparing the years 2016, 2017 and 2018. Adapted from [46].

Software systems are designed, implemented, tested, debugged, analyzed, and maintained by many different developers. All these tasks can be facilitated by using several different visualization techniques. From a historical perspective, software understanding tasks leveraged the use of models and visual notations. An example is the Unified Modeling Language (UML), which has been widely used not only to represent and visualize software systems' structure, behavior, and evolution [13], but also to simplify the process of understanding large-scale architectures [17], and even develop new cloud-based systems [23].

This work addresses the challenges mentioned above by exploring how cloud management can benefit more from a model-based approach, combined with more liveness, leading to the notion of live models at run-time. On the one hand, models would help to abstract the lower-level details by creating and exploiting domain models, in a similar way to UML [26]. On the other hand, more liveness would shorten the feedback loop between the developers and the system under development [1,53], thus helping to reduce management complexity by making it easier to understand quickly what the system is doing or is supposed to do.

To explore the pros and cons of this combination of concepts, we developed the *CloudCity* prototype [32], a live management environment tailored for cloud infrastructures. *CloudCity* aims to offer developers a way to gather continuous feedback about their cloud systems, allowing quick and interactive management of a running cloud system, and therefore ease the process of fault location (usually carried by log analysis) and evolution.

The work here presented extends previous work from the authors in the *Live Software Development* paradigm [1,2,32], delving further into the catalog of visual metaphors for representing cloud infrastructures. It also presents the carried out experiments that evaluate the *CloudCity* solution both in terms of scalability and feasibility as well as the obtained results.

This paper is structured as follows: Sect. 2 provides an overview of the main background concepts of this work and presents some related work. Section 3 gives an overview of our approach, followed by some implementation details. Section 5 explains the validation process using a controlled experiment, along with the discussion of the obtained results, followed by final remarks in Sect. 6.

2 Background and Related Work

To ease the process of understanding and managing complex software, many researchers have investigated different techniques, from high-level abstractions to tool support aiming at improving the programming experience. In the context of this work, we found of high relevance the state-of-the-art on Software Visualization, Model-Driven Engineering, and Live Programming, especially the work more closely related to Cloud Management. In particular, we searched for similarities with previous research results, key features, and ideas that could influence our research.

2.1 Software Visualization

Software visualization is the depiction of software—its structure, behavior, and evolution—and its development process in a visual fashion, leveraging static, interactive and multi-dimensional visual metaphors [13]. Different visualization techniques have been used to ease the understanding of source code, architectural design, use cases, system modules, and more.

Kapec [24] presented a hypergraph-based software visualization system to create a *visual programming environment for software developers*. In this approach, relations between components can be transposed to source code as function calls or class inheritance with visible links between edges, storing information about developers and tasks. As heterogeneous programming environments (*i.e.*, using diverse languages) are a common practice that contributes to software complexity, their approach combines hypergraphs with visual data mining techniques hiding the actual implementation but capturing the call relation.

Lanza et al. [28,29] presented a software visualization technique enriched with metrics information, so-called polymetric views. This approach eases the process of understanding the structure of a software artifact and detects problems in the initial phases of a reverse engineering process. The actual visualization requires: (1) a layout considering the selected entities, relationships, and areas of interest into how they should be sorted and displayed (*e.g.*, a tree layout is better suited for the display of an inheritance hierarchy than a circle layout); (2) a set of metrics extracted from the source code entities, which heavily influence the resulting visualization, being suitable to control the state of a software system during development; and (3) a set of entities that are the parts of the system selected for visualization [29].

Wettel et al. [57] software visualization approach, adopted the urban domain—influenced by the role that civil architecture has on software engineering—as the central metaphor to abstract the different parts of the system. Several similarities can be seen between a city and a software system since both are conceived during a planning phase, in which requirements are the foundation; and then both are built incrementally and require constant maintenance.

Using this city metaphor, Wettel et al. present city elements (*e.g.*, buildings and districts) mapped to software system components (classes and packages respectively). Further, to enhance the visualization, the physical properties of the urban artifacts (*e.g.*, color, and dimensions) reflect attributes of the software components.

The concept of such visualization was implemented in *CodeCity* [57] (Fig. 2). As to validate the feasibility and utility of the approach, an empirical evaluation was carried on in a series of experimental runs spanned over six months. Wettel et al. conclude that for the program comprehension and design quality assessment, the city metaphor enabled the creation of efficient software visualizations. The experiments showed improved correctness 24% of the cases and reduced completion time in 12% over similar state-of-the-practice tools.

Merino et al. extended this vision and brought the concept of virtual reality into the idea of the *CodeCity*, the *CityVR* [38]. In *CityVR*, the city visual

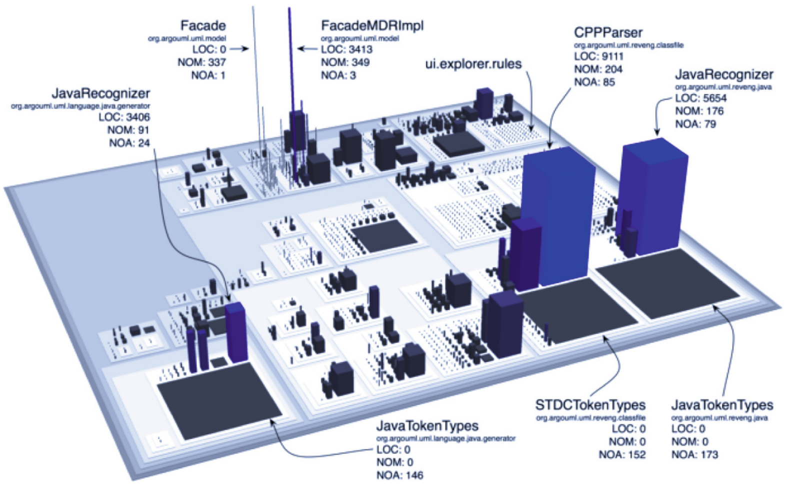


Fig. 2. A 3D representation of the ArgoUML software using the *CodeCity* concept by Wettel et al. [57].

metaphor is enhanced by allowing the developer to explore software pieces in an immersive 3D environment medium.

Other works explore the same idea, such as: (1) *ExplorViz*, a VR approach following the 3D city metaphor [18]; (2) *VR City*, a modification of the city metaphor in virtual reality environment, with a different layout technique that provides a higher level of detail and positioning oriented to the coupling between classes [56]; (3) *SwiftCity*, an application of the City visual metaphor to Swift projects [40]; and (4) Amaral et al. approach for a live development environment for Java using 3D and VR [2].

2.2 Model-Driven Engineering

Models raise the level of abstraction, revealing the big picture, or providing a focus on specific aspects of a system. Model-based approaches have been used as a way to specify the structure and behavior of a system for a long time. UML is one example of a modeling language that is methodology-independent and platform-independent [41]. Although not a visualization approach, but rather a visual notation, it is closely related to software visualization [57].

Sandobalin et al. [47] present ARGON, a solution to help the management of Infrastructure as Code (IaC), through a Domain-Specific Modeling Language (DSML). ARGON is a modeling tool for specifying the final state of the infrastructure and provisioning of cloud resources. The tool aims mainly on the automatic generation of infrastructure provisioning scripts. One of the advantages of this approach is the abstraction from the complexity of working with different cloud providers, resulting in a platform-independent metamodel and thus mitigating the vendor lock-in issue.

Mastelic et al. [33] take advantage of model-driven development for building and managing arbitrary cloud services in a cloud-agnostic manner. The presented CoPS metamodel can describe cloud services using three sequential models: (1) Component, that defines the configuration of each component of the service; (2) Product, that defines the arrangement of the service; and (3) Service, that defines services requirements.

Ardagna et al. [6] defends the same purpose of applying model transformation techniques to instantiate the system into possible multiple clouds. The result aims to be an Integrated Development Environment (IDE) to build and deploy applications in a cloud-agnostic way, adding the concept of multi-clouds.

2.3 Live Programming

As pointed by Sean McDirmid, "programming burdens our minds as we must imagine how the code will execute while editing it" [34,49]. Christopher Hancock [20] in his thesis compares this to archery: aiming an arrow (editing code) involves mentally simulating a physical system while shooting (debugging) provides discrete feedback for the next shot. In other words, to find the cause of errors in software, one should resort to debugging to get feedback about how the code behaves, and this causes a break in the mental flow and the editing process [34].

Live Programming is an idea pioneered by programming environments from the earliest days of computing, such as those for Lisp and Smalltalk. One thing they had in common is *liveness*: an always-available evaluation and nearly instantaneous feedback, usually focused on coding activities. Tanimoto targeted the "edit-compile-link-run" loop, proposing to blur it into a continuum, where the programmer and the system interact in a very tight way—*live* [52,53].

Back to Hancock's analogy, consider hitting a target with a stream of water: we keep correcting our aim until the target is hit, where, unlike archery, we receive continuous feedback on where we are shooting [49].

By unifying the gap between code editing and debugging [49], re-executing the program and providing continuous feedback while editing eases the burden of programming [34]. It is not a *silver-bullet* for software systems development, but potentially very important for some. While the ability to inspect and modify is taken for granted in most IDEs, adding liveness is an enhancement [53].

Examples of liveness can be observed in several IDEs that already provide continuous and responsive feedback on the lexical, syntactic, and type safety of the developer's code. Further, many *live* visual programming languages such as VIVA, Forms/3, Morphic, and PureData go beyond this by providing live feedback about how the program executes as the code is edited [34].

Some challenges to this concept have been pointed out on how feedback may be considered harmful, since that receiving continuous results with change can be potentially distracting in some cases, forcing the programmer to write in a particular order to *keep it live*. For live programming to succeed, it must enhance programming without restricting what the programmer can do, either beginner or expert [35].

Other frequent critics highlight the fact that the steps in between execution are the essential part of programming, and, the usage of *liveness* can result in hiding some critical parts of the flow of execution, with the developer only focusing on the program output [34]. Nevertheless, from a debugging perspective, live programming can address this concern combining editing and debugging, having debug results readily visible while editing, thus returning the focus to the program flow and how changes affect specific parts of execution [34].

Although the notion of *Live Programming* focuses on the particular activity of *programming*, there is nothing in its principles that cannot be applied to many other activities, such as: requirements analysis, design, testing, deployment, or maintenance. Therefore, Live Software Development concerns on achieving higher *liveness* in more development activities beyond programming [1].

2.4 Cloud Management

The trend has been to leverage clouds as complex, highly heterogeneous, and distributed architectures, including hybrid and multiclouds [31]. This growth has given rise to new challenges and technologies to deal with them, namely with governance, security, and management.

The process of obtaining services from the cloud, such as spawning computers or virtual hosts and tailoring its software and configurations, is known as *provisioning* [9]. Inspite of its close relation to deployment of services or applications, provisioning does not necessarily imply new deployments or *vice versa* [48].

The widespread use of cloud computing has been empowering the movement of DevOps—a software engineering culture aiming to unify software development (oriented to change) and software operation (oriented to stability) [4,15] – due to its benefits when comparing to traditional operations processes, namely:

Rapid Delivery. Quickly respond to customer needs and move a change into production [4,15].

Reliability. Ensure the quality of application updates and infrastructure changes through testing in practices such as continuous integration and continuous delivery (CI/CD) [4,15].

Scale. Automation and consistency help changing systems efficiently and with reduced risk [4].

Collaboration. Developers and operation engineers share responsibilities and combine workflows [4].

This movement has increased the responsibilities of developers beyond *programming*, having now an increasing role in the building, continuous integration, and fast delivery (building an effective pipeline of releases) of new services and applications. Thus, developers now need to focus more on the configuration management (*e.g.*, cloud configuration management), testing, and production of these systems [27].

Configuration Management. Configuration Management (CM) is a core part of the *provisioning* process, methodically handling changes to a system to maintain its integrity over time. Pressman et al. define such process as [45]:

> "A set of activities designed to manage change by identifying the work products that are likely to change, establishing relationships among them, defining mechanisms for managing different versions of these work products, controlling the changes imposed, and auditing and reporting on the changes made."

One can identify *automation*—the ability to automatically deploy new system versions in the existent infrastructure—as the most fundamental concept in configuration management. Thus, commonly, configuration management tools are presented as Automation Tools or IT Automation Tools [22]. Examples of CM tools include Chef and Puppet [5,46,58]. Both allow the specification of infrastructure as code (so-called *recipes*) by using a domain specific language.

Although these approaches spread a notion of controlled and reliable mutable oriented node configuration, it is important to consider the possibility of configuration drifts, over time, as each server builds a unique history of changes [8].

Infrastructure Orchestration. The main feature of configuration management tools is to install software on resources that already exist. Orchestration *per se* has a different purpose than CM. Orchestration tools are typically designed to enforce a particular workflow order to a set of automated tasks, such as the provisioning of those resources. However, both orchestration and CM categories are not mutually exclusive, with some orchestration tools extending its features to configuration and vice-versa [8].

Examples of Infrastructure Orchestration tools are CloudFormation, the AWS-based orchestration tool to describe and provision infrastructure as code, and Terraform, a similar tool but cloud-agnostic, enabling the combination of multiple cloud service providers with a unified syntax [21]. Both tools focus on the definition of a blueprint for controlling and versioning resources configurations easily, which typically defaults to an immutable infrastructure paradigm.

As a summary, we conclude that most of the approaches and tools for managing cloud services prevail on the concept of infrastructure as code, with some following a kind of model-driven approach, namely to manage multiple cloud services and thus to avoid vendor lock-in. However, the existing tools only provide minimal, or even none, live support, one of the aspects we focus on exploring with this work.

3 CloudCity: The Approach

Resulting from the lack of resources and expertise on how to handle and manage different cloud services altogether [46], there has been an increasing interest in novel approaches to support infrastructure provisioning, orchestration and configuration management. Most of these approaches have requirements of automation and orchestration, due to the ever-growing complexity and scale of systems (*e.g.*, Internet-of-Things) [44,54].

3.1 Overview

To tackle the current issues in cloud computing, while taking into account the existent requirements, our tool, named *CloudCity*, uses a 3D visualization approach for managing cloud infrastructures. The chosen visual metaphor, the city metaphor, was based on the work by Wettel et al. [57] in *CodeCity 3D* since the software engineering scientific community already validated it with good empirical results in what regards software visualization. Also, using a city to represent an infrastructure intends to help the user familiarizing with the domain by using already known city objects.

The need for metaphors arises from the fact that the cloud is not a physical entity. Thus, by nature, it cannot be purely synthesized into a straightforward, visually understandable mapping. However, it can be transposed into other dimensions, such as code (*c.f.* Terraform and CloudFormation) or models as a way to ease the process of managing such infrastructures. Representing clouds with a validated metaphor, the city, enables users to gradually become familiar with the described architecture, due to the many similarities between the two domains.

CloudCity embraces the concept of *liveness*, underlying Live Software Development [1], allowing the developer to get continuous feedback on *how architectural* (instead of code) *changes affect the whole system*, going beyond a static 3D visualization of a cloud architecture.

In detail, the main objective of *CloudCity* is to allow the design and analysis of cloud compositions through a mostly-intuitive mapping between city objects (*i.e.*, houses, streets, skyscrapers) and cloud resources. Each one of the buildings contains a set of properties reflected from the cloud, which can be inspected or modified through simple user interaction. Relations between elements are depicted as curved lines between them, which can be filtered and inspected on-demand. The main difference from other model-driven approaches is that this environment does not reflect a static infrastructure mapping, but instead a live infrastructure showing the real-time state of each component—a metaphor that we introduce as, *the live city.* An example of *CloudCity*'s main environment is depicted in Fig. 3.

Regarding the tools provided to the developer, the user interface is composed of the interactive components listed below. All the panels are collapsible, triggered by a user action, to save visual space.

Information Panel. Acts as an inspector with information about the selected resource.

Resource Context Menu. Acts as a dynamic options menu, with several actions depending on the selected resource.

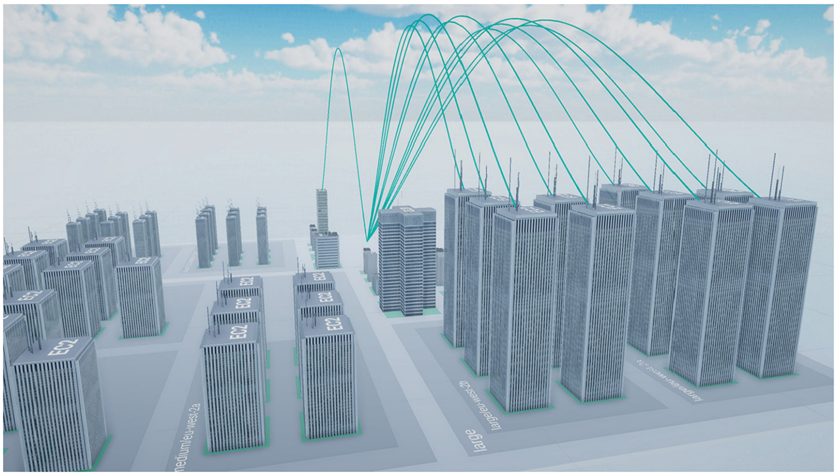


Fig. 3. CloudCity's main environment containing an small size example architecture, displaying all relations.

Plane Main Menu. Contains global infrastructure actions, such as spawning new resources.

Regarding the environment's background, it consists of a simple skybox chosen to increase the resemblance to a city's atmosphere.

3.2 Architecture

CloudCity follows a model-driven engineering philosophy, embracing the concept of models as a way to express the system and the relation between system parts.

CloudCity high-level architecture, depicted in Fig. 4, is composed of three core components, namely:

Cloud Service Providers API. Provides a connection to a specific cloud service provider, thus allowing to fetch and interact with the cloud architecture.
Importer. Periodically pools or checks the provider and detects changes in the infrastructure state, forwarding actions to update specific resources.
Resources. The elements correspond to different cloud services. These resources follow a composite pattern, *viz.* a group of resources can either contain a resource or another resource group. If it contains another group, the same applies recursively downwards the tree structure.

3.3 Proof-of-Concept

For the sake of simplicity, some technological decisions were made to ease the development of a proof-of-concept. In what regards Cloud Service Providers API integration, we focused only on Amazon Web Services among the existent

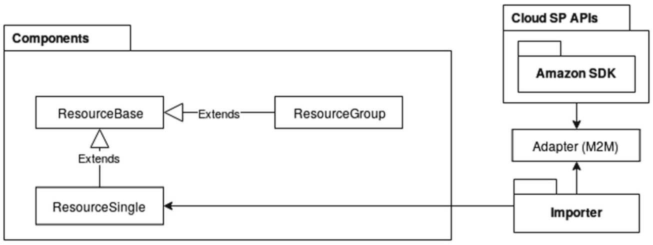


Fig. 4. CloudCity's architecture described in a package diagram [32].

options. Regarding the fetch of information from the cloud provider, we implemented a pooling approach instead of a more efficient one, such as event-driven, or publish-subscribe approach, due to limitations of the API of the provider itself.

However, even given the fact that the proof-of-concept integrates with only one cloud provider, the system is built in a modular way that allows the addition of new adapters to different cloud services providers, easing the process of integrating with other sellers such as Microsoft Azure. This capability is accomplished by dividing the *CloudCity* architecture into two decoupled layers:

Platform Independent Model. Illustrated in Fig. 5, this model is independent of any specific provider.

Platform Specific Model. This model is coupled with a specific provider and can be obtained with a model to model transformation.

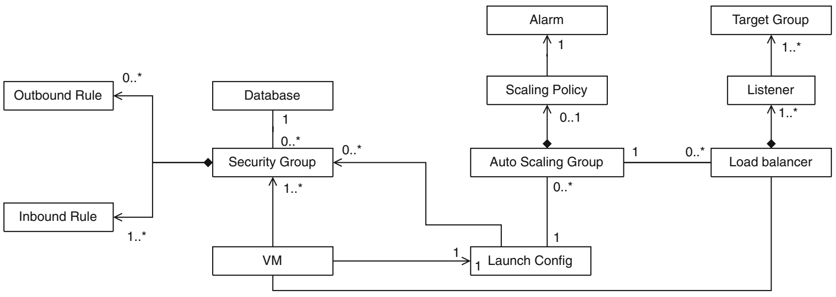


Fig. 5. Infrastructure metamodel (inspired by the abstract syntax presented by Sandobalin et al. [47]) [32].

The proof of concept was implemented using a multipurpose three-dimensional engine, Unity. This solution also opens doors for new features to be studied and provides support for Virtual/Augmented Reality, an exciting perspective also studied in other approaches using the City Metaphor [38,56].

4 CloudCity: The Live Environment

One of the key challenges was to find the most suitable abstractions—within the urban catalog of metaphors—to portray the cloud architecture infrastructure, while still being compatible with *live* features and easy to use and understand. To be able to achieve this, we decided to start by using the following metaphors:

Resource Mapping. Establishing a correspondence between the resources offered by cloud providers and the catalog of metaphors available (or permutations of those metaphors).

Layout. Defining a proper environment that lays out the different components in an understandable way and adjusts automatically as the cloud architecture is modified while remaining consistent throughout the process.

Updates and Interactions. Support the *live* aspects of the environment, *i.e.*, how to translate the infrastructure updates into a human-understandable notation in real-time.

The following subsections describe these three aspects in more detail.

4.1 Resource Mapping

The number of services offered by cloud providers is continuously growing. This growth is mostly driven by the necessity of providers to adapt their offer to clients, to maintain their position in a demanding market [51].

However, at any given point in time, there is a finite set of services and resources with properties known *a priori*. This fact allows the creation of an *alphabet*, which can be expanded accordingly to new services that can appear, with models for each one of the elements that need to be represented, rather than defining new models on-the-fly.

Even so, due to the current number of services in the portfolio of the cloud providers, we focused on creating models only for the most common and popular services across cloud providers [30]. The following list describes those services along with the respective model and their urban-based visual metaphor.

Security Group. Virtual firewalls to control instances (*e.g.*, virtual machines) inbound and outbound traffic. Each security group contains a set of rules which control the port range where traffic is allowed. The metric chosen for the building height varies according to the port range the security group covers. Due to the commonality of this element in cloud architectures (a VM instance can have from one up to five different groups), the building dimensions correspond to the *small* building type as depicted in Fig. 6, resulting in the Fig. 7b.

Virtual Machine. VMs are one of the most common elements in cloud computing since they provide scalable computation capacity in the cloud. Each instance has a pre-determined size depending on its hardware specifications. The metric for the building dimensions varies according to this attribute accordingly with Fig. 6, and their visual representation is given in Fig. 7a.

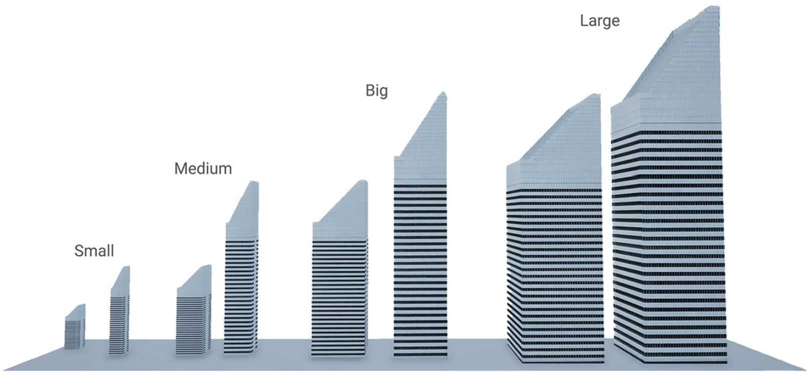
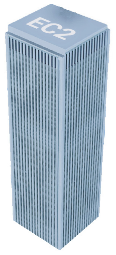
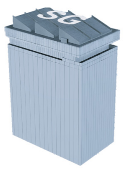


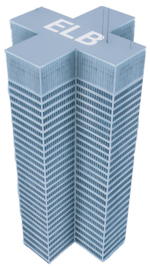
Fig. 6. CloudCity's reference building dimensions sorted in ascending order [32]. Nano VM instances are considered *small*, Micro and Small are considered *medium*, from Large to 8x Large are considered *big* and from 8x Large to 32x Large are considered *large*.



(a) Metaphor for a Virtual Machine (VM).



(b) Metaphor for a Security Group.

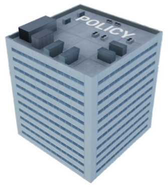


(c) Metaphor for a *Elastic* Load Balancer

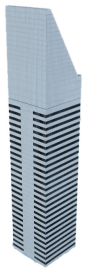
Fig. 7. Visual notation for a Virtual Machine, Security Group and Load Balancer [32].

Load Balancer. Element that distributes traffic across multiple targets for achieving multi-tenancy and resource pooling. A Load Balancer can have multiple listeners that receive incoming connections and distribute them across multiple groups of targets. The building size fluctuates depending on the total number of rules that the load balancer takes into consideration when forwarding connections, depicted in Fig. 7c. It is part of the *big* buildings category since it is a central component between the point of entry and the targets.

Scaling Policy. Policies define how the scaling group increases or decreases the size, and according to which metrics. The building height varies depending on the scaling adjustment, Fig. 8a, and the building type falls in the *medium* buildings category since it can be considered a subset of the Auto Scaling Group.



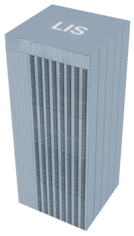
(a) Metaphor for a Scaling Policy.



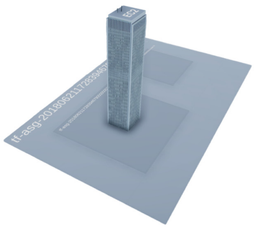
(b) Metaphor for a Target Group.

LAUNCH CONFIG

(c) Metaphor for a Launch Configuration.



(d) Metaphor for a Listener.



(e) Metaphor for an Auto Scaling Group with a VM.

Fig. 8. Visual notation for a Scaling Policy, Target Group, Launch Configuration, Listener and Auto Scaling Group.

Target Group. A target group routes incoming listener requests to one or more registered targets. The building height varies depending on the number of instances registered in it, Fig. 8b. It is considered as part of the *medium* buildings category since this component can also be considered a subset of the load balancer.

Launch Configuration. This is a parental reference of machine specifications for a VM to be mirrored from, guiding the Auto Scaling Group as it expands the number of replicated instances. The building type chosen for this component depends on the instance type attribute (*c.f.* Fig. 6), and it is represented in Fig. 8c.

Listener. Listeners are responsible for checking for incoming requests on a specific port and forward them to a Target Group. The building height varies in consonance with the number of rules it takes consideration when forwarding a connection to a specific group of targets, Fig. 8d. The building is part of the *medium* buildings category.

Auto Scaling Group. This element is not depicted as a building since it is a group with multiple VMs and scales dynamically. The metaphor chosen was a plane with sufficient area to support the different availability zones and specific VM, visually represented on Fig. 8e.

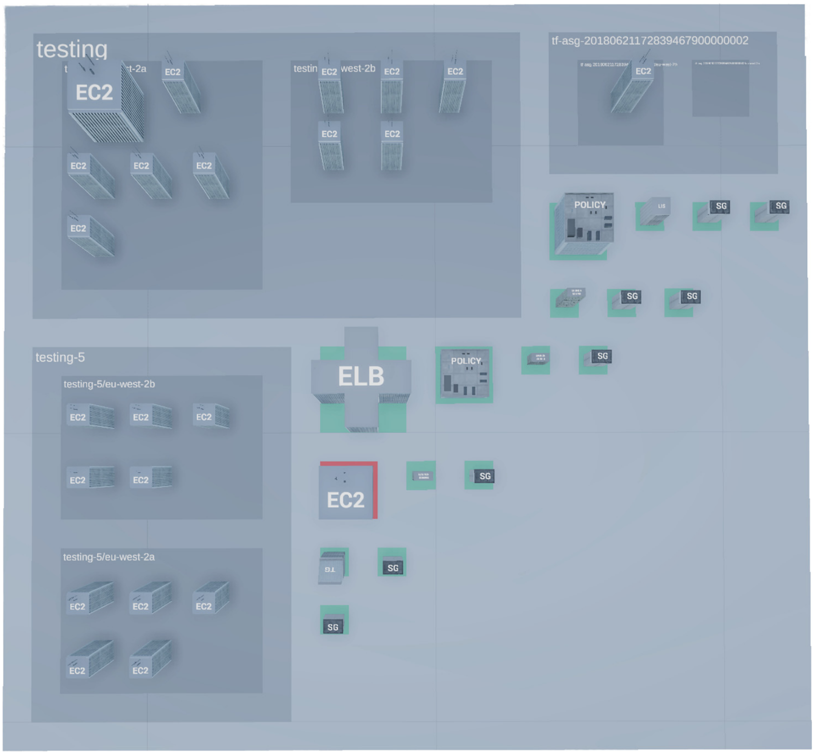


Fig. 9. An example of the rectangle packing layout for a considerable size infrastructure, composed of: three Auto Scaling Groups containing multiple size instances and two Scaling Policies; a stopped Virtual Machine; one Load Balancer and several Security Groups [32].

4.2 Layout

To be able to manage a cloud infrastructure in a live way, there is the need for a mechanism to layout and update components quickly in the tool's environment, as the architecture expands and is modified. It has to: (1) support laying out all the imported components of the infrastructure, with different dimensions, in an ordered and understandable manner; (2) optimize the number of buildings, not wasting much of the cities' real-estate [57]; (3) support grouping components according to a class.

The strategy picked for layering the elements is *CodeCity*'s rectangle packing algorithm proposed by Wettel et al. [57]. This approach starts with an empty rectangular space, large enough to host a set of exposed components. In each step, the elements are laid out in the best free space from a list of potential candidates. In case the element does not cover the full space, we recursively split the surplus in two different cuts available to host new components, as depicted in Fig. 9.

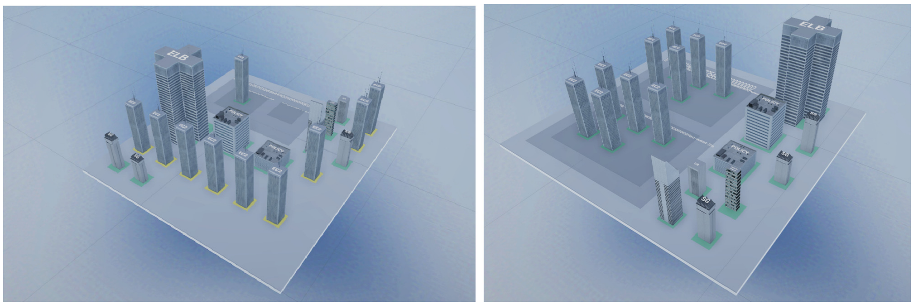
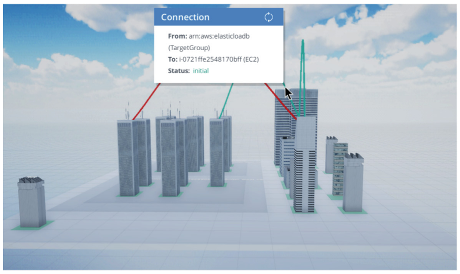


Fig. 10. Visualization of an infrastructural update when an auto-scaling group enters in action, scaling the number of instances from 1 to 10. As result nine new servers were spawned (left) and then attached inside the scaling group (right) [32].

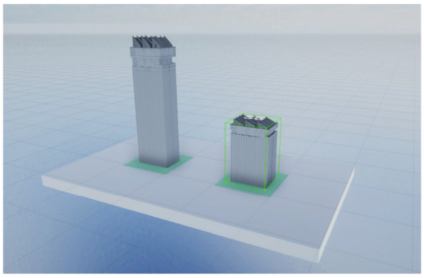
4.3 Updates and Interactions

To be able to fetch information about the cloud infrastructure, as well as any subsequent updates, a pooling approach was implemented, that checks for differences between each response. Initially, we planned to use a publish-subscribe pattern, but due to some limitations by the provider, we had to settle with a poll mechanism.


Connection

From: arn:aws:elasticloadb (TargetGroup)
To: i-0721ffe2548170bff (EC2)
Status: initial

(a) Helper window that allows the inspection of links between elements.



(b) Selection of an element within the environment.

Fig. 11. Representation of some user interactions within the live environment.

Having communication-enabled and a method to detect the change, the next step is to refresh the infrastructure when changes happen. The most naive approach would be to destroy the whole infrastructure and rebuilt it. However, for efficiency reasons, we decided not to destroy any element except if it has been terminated. Instead, every time the layout needs to re-position elements, only the affected ones change position, as depicted in Fig. 10.

To avoid abrupt changes in the layout, all components change their position *slowly* (speed of 1 unit per second) to increase the response feedback (sliding in-between positions), making it easier for the developer to understand changes.

Relations are mapped as arcs beginning at one instance, or group, and ending in another. Both resources and their relations may contain a state depending on their nature; which can be inspected by clicking it, and filtered when a specific component is selected, as depicted in Fig. 11a. Cloud elements, represented as different buildings, can be selected (Fig. 11b) and configured with the aid of windows within the 3D environment.

5 Experiments and Results

There is a broad consensus in the software visualization community, and also in the broader information visualization community, that a lack of proper evaluation that can demonstrate the effectiveness of tools is detrimental to the development of the field [50].

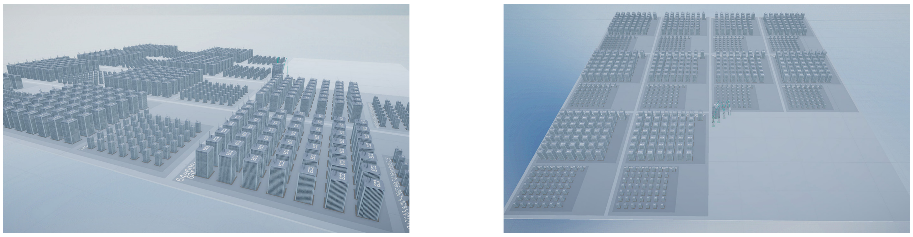


Fig. 12. Different views of the stress test with 10 Auto Scaling Groups and 1000 Virtual Machines.

5.1 Sanity Checks

To test the visualization of a considerable sized infrastructure, we simulated an environment composed of 10 different Auto Scaling Groups and a total of 1000 servers, as in Fig. 12.

We concluded that having a large number of resources together in a unique model eventually becomes unnecessary and inefficient for considerably large infrastructures. Conversely, if we divided or collapsed large groups of resources by their Auto Scaling Group, availability zone, or even resource type, we would accomplish a higher-level analysis of a cloud architecture. Thus, avoiding updates in locations far away from our focus zone.

5.2 Controlled Experiment

We designed a controlled experiment to assert the feasibility of *CloudCity*. In this experiment, we focused on: (1) creating and managing a collection of related AWS resources; and (2) inspecting a running architecture and update it on-the-fly.

The population under survey consisted of 18 MSc students, ranging from those with experience in cloud computing to those with little or no knowledge of it. The experiment consisted in performing a similar set of tasks using three different tools. The goal was to evaluate the effect of the tools on the completion of the tasks.

The controlled experiment was designed to probe different perspectives, which were combined into two distinct phases: construction and analysis.

One of the objectives is to compare the feasibility of CloudCity when comparing with state-of-the-practice tools, namely AWS CloudFormation, which allows the specification of architecture in a blueprint file, providing a static visualization of it. However, AWS CloudFormation does not give the ability to inspect a running architecture, and, as such, an additional tool was used for fulfilling this aspect, namely AWS Management Console.

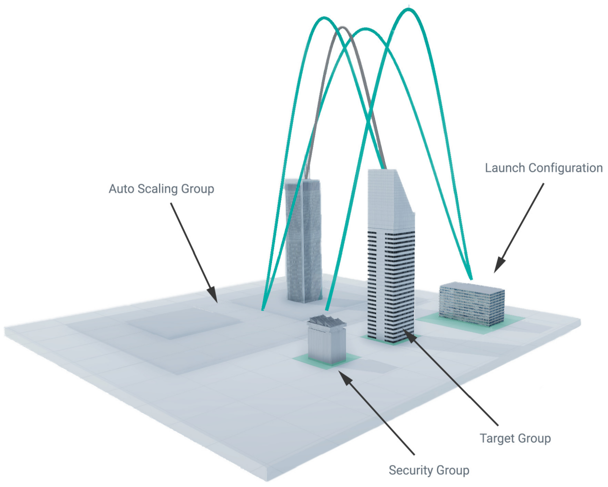
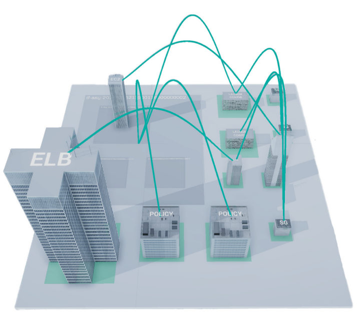


Fig. 13. An illustration of the resulting CloudCity model in the construction phase [32].

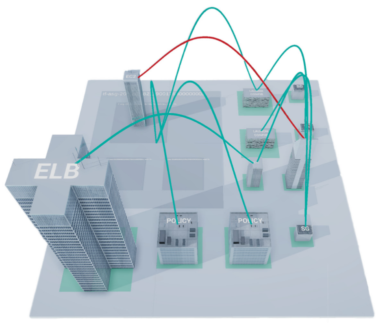
Although the data from the experiments are too scarce for definite and sound conclusions, we could assert the feasibility of the solution, and the experimental treatment represents a considerable improvement over some of the current practices, appointing the open potential of adding *liveness* to current cloud management tools.

Construction Phase. In the construction phase, the participants were asked to design a simple cloud architecture composed of four resources using CloudCity, namely: (1) an Auto Scaling Group with a minimum size of two instances; (2) a Launch Configuration for each new instance to be spawned inside the scaling group; (3) a Security Group; and (4) a Target Group to route incoming requests to the targets in the scaling group. The expected, resulting model is depicted in Fig. 13.

All of the participants were capable of fulfilling this task by using the provided *alphabet* and the environment, initially, identifying the right resources and then proceeding to configure and connect them according to the requirements.



(a) The resulting model of the second phase of the experiment according to Terraform's configuration plan.



(b) The result after the misconfiguration of a Security Group, for the purpose of identifying the resulting infrastructural changes (unhealthy VM instance).

Fig. 14. Resulting visualizations of the experiments. The connections between elements are representing connections and the floor gives information about the group/context of those elements. If everything is operating normally both the connections and floor colors are in green, otherwise the connections and/or floor colors of each element turn red (Color figure online) [32].

Analysis Phase. As for the analysis phase, it consisted of inspecting an existing infrastructure. In order to keep the experience randomized and create some independence between the two phases of the experiment, we previously prepared a similar infrastructure using Terraform containing: (1) an Auto Scaling Group in two zones connected to the respective launch configuration; (2) two Scaling Policies; (3) a Load Balancer with respective Listener, Target Group and Security Group; and (4) a simple HTTP web service running on port 80 (Virtual Machine).

The rationale of the second phase was to simulate the occurrence of an unhealthy target, a common event in a cloud environment. In most cases, the cause is due to a failed/overloaded VM instance or Security Group misconfiguration. For that purpose, we misconfigured a Security Group (firewall) on purpose in one of the registered targets and disallowed any traffic coming from the Target Group. In consequence, the Target Group was not able to send health check requests, and consider the instance unhealthy. The goal is to locate that specific instance and analyze its cause, targeting *liveness* level three: informative, significant, and responsive [53]. Both the occurrences can be confirmed in Fig. 14a and b.

All of the participants were able to identify the issue by observing the red connection (failed health check) between the Target Group and the Virtual Machine instance. They were able to inspect it (by clicking on the connector) and, by tracing back the origin of the problem to the Security Group, they were able to create a new rule to allow the traffic.

6 Final Remarks

There are several issues with cloud management resulting from: (1) the cloud providers being always developing new services to keep up with a demanding market and as reaction to new paradigms (*e.g.*, IoT), and (2) the unavoidable increasing complexity when too many resources are under management in an overwhelming disheveled environment.

From the viewpoint of cloud management, the main contribution of this work is a development environment for cloud architectures, *i.e.*, an approach to analyze, architect and configure cloud compositions with a higher level of abstraction. This environment allows developers to focus more on their business logic and track the changes as the infrastructure evolves, and its complexity increases.

The *CloudCity* approach, resulting from a combination of strengths from several tools and methods for developing cloud architectures and software in general, explores the concept of Live Software Development [1] in the cloud domain, by shortening the feedback loop between the developer and the infrastructure, allowing them to quickly understand, almost immediately, how the infrastructure reacts to change.

As per the comparison to the current state-of-the-practice, we consider that increasing *liveness* improves the developers' experience in cloud architecture configuration tasks. The carried controlled experiment asserted the feasibility and sanity (*i.e.*, evaluate if the approach works as the cloud architecture scales) of the *CloudCity* approach, although further validation is needed to assert aspects such as the *efficiency*—achieving the results in a faster way compared to the traditional methods doing the same task—and overall developer experience. An empirical validation within an industrial case scenario would bring useful information about the *usefulness* of the approach.

During the development of this approach several future research directions where uncovered, such as (1) providing a modifiable layout technique—a user's ability to manually modify the position of a specific component; (2) explore other levels of liveness following the Tanimoto 6-level scale [53]; (3) investigate different metaphors beyond the one of Wettel et al. [57]; and (4) adding other services offered by cloud providers to the *alphabet* (*e.g.*, dealing with the new services related to IoT would bring new challenges such as how to deal with a mixture of virtual infrastructure and real infrastructure, *i.e.*, gateways, sensors, and actuators [12]).

References

1. Aguiar, A., Restivo, A., Figueiredo Correia, F., Ferreira, H.S., Dias, J.P.: Live software development: tightening the feedback loops. In: Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming. Programming 2019 Companion (2019)
2. Amaral, D., Domingues, G., Dias, J.P., Ferreira, H.S., Aguiar, A., Nóbrega, R.: Live software development environment for Java using virtual reality. In: Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering. ENASE, vol. 1, pp. 37–46 (2019)
3. Amazon, A.: Announcing Amazon Elastic Compute Cloud (Amazon EC2) - beta (2006). https://aws.amazon.com/about-aws/whats-new/2006/08/24/announcing-amazon-elastic-compute-cloud-amazon-ec2---beta/. Accessed 07 2019
4. Amazon Web Services: what is devops? (2017). https://aws.amazon.com/pt/devops/what-is-devops/
5. Anicas, M.: Getting started with puppet code: manifests and modules (2014). https://www.digitalocean.com/community/tutorials/getting-started-with-puppet-code-manifests-and-modules
6. Ardagna, D., et al.: MODA CLOUDS: a model-driven approach for the design and execution of applications on multiple clouds. In: Modeling in Software Engineering, pp. 50–56 (2012)
7. Armbrust, M., Fox, A., Griffith, R., Joseph, A., Katz, R.H.: Above the clouds: a Berkeley view of cloud computing. Technical report, University of California, Berkeley, UCB, p. 1 (2009)
8. Brikman, Y.: Why we use Terraform and not Chef, Puppet, Ansible, SaltStack, or CloudFormation (2016). https://blog.gruntwork.io/
9. Buyya, R., Broberg, J., Goscinski, A.: Cloud Computing Principles and Paradigms. Wiley, Hoboken (2011)
10. Cachin, C., Schunter, M.: A cloud you can trust. IEEE Spectr. **48**(12), 28–51 (2011)
11. Microsoft Corporation: Microsoft Azure IoT Reference Architecture. Technical report, Microsoft Corporation (2016). https://azure.microsoft.com/de-de/updates/microsoft-azure-iot-reference-architecture-available/
12. Dias, J.P., Faria, J.P., Ferreira, H.S.: A reactive and model-based approach for developing internet-of-things systems. In: 2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC), pp. 276–281, September 2018
13. Diehl, S.: Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-46505-8
14. Dillon, T., Wu, C., Chang, E.: Cloud computing: issues and challenges. In: 2010 24th IEEE International Conference on Advanced Information Networking and Applications, pp. 27–33, April 2010

15. Edwards, D.: What is devops? (2010). http://dev2ops.org/2010/02/what-is-devops/
16. Erian, T.E.: The XaaS family: understanding IaaS, PaaS and SaaS (2018). https://www.ibm.com/blogs/cloud-computing/2014/10/31/xaas-family-iaas-paas-saas-explained/
17. Fittkau, F., Waller, J., Wulf, C., Hasselbring, W.: Live trace visualization for comprehending large software landscapes: the explorviz approach. In: 2013 First IEEE Working Conference on Software Visualization (VISSOFT), pp. 1–4, September 2013
18. Fittkau, F., Krause, A., Hasselbring, W.: Exploring software cities in virtual reality. In: 2015 IEEE 3rd Working Conference on Software Visualization, VISSOFT 2015 - Proceedings, pp. 130–134 (2015)
19. Fraser, S.D., et al.: No silver bullet reloaded: retrospective on essence and accidents of software engineering. In: Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion, pp. 1026–1030. ACM (2007)
20. Hancock, C.M.: Real-time programming and the big ideas of computational literacy. Ph.D. thesis, Massachusetts Institute of Technology (2003)
21. HashiCorp: Terraform vs. other software (2017). https://www.terraform.io/intro/vs/index.html
22. Heidi, E.: An introduction to configuration management (2016). https://www.digitalocean.com/community/tutorials/an-introduction-to-configuration-management
23. Junior, F.M.R., da Rocha, T.: Model-based approach to automatic software deployment in cloud. In: CLOSER, pp. 151–157 (2014)
24. Kapec, P.: Visualizing software artifacts using hypergraphs. In: Proceedings of the 26th Spring Conference on Computer Graphics - SCCG 2010, p. 27 (2010)
25. Kavis, M.J., et al.: Architecting the Cloud: Design Decisions for CloudComputing Service Models (SaaS, PaaS, and IaaS). Wiley, Hoboken (2013)
26. Kent, S.: Model driven engineering. In: Butler, M., Petre, L., Sere, K. (eds.) IFM 2002. Lecture Notes in Computer Science, vol. 2335, pp. 286–298. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-47884-1_16
27. Kerzazi, N., Adams, B.: Who needs release and devops engineers, and why? In: Proceedings of the International Workshop on Continuous Software Evolution and Delivery - CSED 2016, pp. 77–83 (2016)
28. Lanza, M.: CodeCrawler - Polymetric views in action. In: Proceedings - 19th International Conference on Automated Software Engineering, ASE 2004, pp. 394–395 (2004)
29. Lanza, M., Ducasse, S.: Polymetric views-a lightweight visual approach to reverse engineering. Trans. Softw. Eng. (TSE) **29**(9), 782–795 (2003)
30. Li, A., Yang, X., Kandula, S., Zhang, M.: CloudCmp: comparing public cloud providers. In: Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement, IMC 2010, pp. 1–14. ACM, New York (2010)
31. Linthicum, D.S.: Understanding complex cloud patterns. IEEE Cloud Comput. **3**(1), 8–11 (2016)
32. LourenÇo, P., Dias, J.P., Aguiar, A., Ferreira, H.S.: CloudCity: a live environment for the management of cloud infrastructures. In: Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering. ENASE, vol. 1, pp. 27–36 (2019)

33. Mastelic, T., Brandic, I., Garcia, A.G.: Towards uniform management of cloud services by applying model-driven development. In: 2014 IEEE 38th Annual Computer Software and Applications Conference, pp. 129–138 (2014)
34. McDirmid, S.: Usable live programming. In: Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, Onward! 2013, pp. 53–62. ACM, New York (2013)
35. Mcdirmid, S.: The promise of live programming. In: LIVE Programming Workshop (2016)
36. Mell, P., Grance, T.: The NIST definition of cloud computing recommendations of the national institute of standards and technology. Technical report, NIST (2011)
37. Mens, T.: On the complexity of software systems. Computer **45**(8), 79–81 (2012)
38. Merino, L., Ghafari, M., Anslow, C., Nierstrasz, O.: CityVR: gameful software visualization. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 633–637, September 2017
39. Janakiram, M.S.V.: AWS service sprawl starts to hurt the cloud ecosystem (2018). https://www.forbes.com/sites/janakirammsv/2018/01/08/aws-service-sprawl-starts-to-hurt-the-cloud-ecosystem/#44616e775c1f. Acceseed July 2019
40. Nunes, R., Reboucas, M., Soares-Neto, F., Castor, F.: Visualizing swift projects as cities. In: Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering Companion, ICSE-C 2017, pp. 368–370 (2017)
41. Object Management Group, Inc.: Introduction to OMG's unified modeling language (2005). http://www.uml.org/what-is-uml.htm
42. Oladehin, O., Brett, F.: Core tenets of IoT. Technical report, Amazon Web Services (2017). https://d1.awsstatic.com/whitepapers/core-tenets-of-iot1.pdf
43. Opara-Martins, J., Sahandi, R., Tian, F.: Critical review of vendor lock-in and its impact on adoption of cloud computing. In: International Conference on Information Society (i-Society 2014), pp. 92–97. IEEE (2014)
44. Pinto, D., Dias, J.P., Sereno Ferreira, H.: Dynamic allocation of serverless functions in IoT environments. In: 2018 IEEE 16th International Conference on Embedded and Ubiquitous Computing (EUC), pp. 1–8, October 2018
45. Pressman, R.S., Maxim, B.R.: Software Engineering: A Practitioner's Approach. McGraw-Hill Education, New York (2015)
46. RightScale: State of the Cloud Report. Technical report, RightScale (2017)
47. Sandobalin, J., Insfran, E., Abrahao, S.: An infrastructure modelling tool for cloud provisioning. In: Proceedings - 2017 IEEE 14th International Conference on Services Computing, SCC 2017, pp. 354–361 (2017)
48. Sayers, D.: Configuration management vs. application release automation (2017). https://devops.com/configuration-management-vs-application-release-automation/
49. McDirmid, S.: Live programming as gradual abstraction. In: LIVE Programming Workshop (2017)
50. Sensalire, M., Ogao, P., Telea, A.: Evaluation of software visualization tools: lessons learned. In: 2009 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis, pp. 19–26 (2009)
51. Serrano, N., Gallardo, G., Hernantes, J.: Infrastructure as a service and cloud technologies. IEEE Softw. **32**, 30–36 (2015)
52. Tanimoto, S.L.: VIVA: a visual language for image processing. J. Vis. Lang. Comput. **1**, 127–139 (1990)
53. Tanimoto, S.L.: A perspective on the evolution of live programming. In: 2013 1st International Workshop on Live Programming, LIVE 2013 - Proceedings, pp. 31–34 (2013)

54. Tosatto, A., Ruiu, P., Attanasio, A.: Container-based orchestration in cloud: state of the art and challenges. In: Proceedings - 2015 9th International Conference on Complex, Intelligent, and Software Intensive Systems, CISIS 2015, pp. 70–75 (2015)
55. Vaquero, L.M., Rodero-Merino, L., Caceres, J., Lindner, M.: A break in the clouds: towards a cloud definition. ACM SIGCOMM Comput. Commun. Rev. **39**(1), 50–55 (2008)
56. Vincur, J., Navrat, P., Polasek, I.: VR City: software analysis in virtual reality environment. In: 2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C), pp. 509–516 (2017)
57. Wettel, R., Lanza, M., Robbes, R.: Software systems as cities. In: Proceeding of the 33rd International Conference on Software Engineering - ICSE 2011 (2011)
58. Wettinger, J., et al.: Integrating configuration management with model-driven cloud management based on TOSCA. In: CLOSER 2013 - Proceedings of the 3rd International Conference on Cloud Computing and Services Science, pp. 437–446 (2013)