IEEE CLOUD COMPUTING

VOLUME 5, NUMBER 3 MAY/JUNE 2018

Cloud Reliability

IEEE IEEE computer society IEEE ComSoc IEEE Communications Society

www.computer.org/cloud

IEEE CLOUD COMPUTING

www.computer.org/cloudcomputing

EDITORIAL OFFICE
Publications Coordinator:
cloudreview@allenpress.com
Authors: www.computer.org/web/peer-review/magazines
Letters to the Editors: bbrannon@computer.org
Subscribe: www.computer.org/subscribe
Subscription change of address:
address.change@ieee.org
Missing or damaged copies:
help@computer.org
Reprints of articles: cloud@computer.org
IEEE Cloud Computing
c/o IEEE Computer Society
10662 Los Vaqueros Circle,
Los Alamitos, CA 90720 USA
Phone +1 714 821 8380; Fax +1 714 821 4010
www.computer.org/cloud-computing

IEEE ComSoc
IEEE Communications Society

SUSTAINABLE FORESTRY INITIATIVE Certified Sourcing www.sfiprogram.org SFI-01681

IEEE Cloud Computing: (ISSN 2325-6095) is published bimonthly by the IEEE Computer Society. IEEE headquarters: Three Park Ave., 17th Floor, New York, NY 10016-5997. IEEE Computer Society Publications Office: 10662 Los Vaqueros Cir., Los Alamitos, CA 90720; +1 714 821 8380; fax +1 714 821 4010. IEEE Computer Society headquarters: 2001 L St., Ste. 700, Washington, DC 20036. Subscribe: Go to www.computer.org/subscribe for more information on subscribing.

2019

IEEE-CS
Charles Babbage Award

CALL FOR AWARD NOMINATIONS
Deadline 1 October 2018

ABOUT THE IEEE-CS CHARLES BABBAGE AWARD
Established in memory of Charles Babbage in recognition of significant contributions in the field of parallel computation. The candidate would have made an outstanding, innovative contribution or contributions to parallel computation. It is hoped, but not required, that the winner will have also contributed to the parallel computation community through teaching, mentoring, or community service.

ABOUT CHARLES BABBAGE
Charles Babbage, an English mathematician, philosopher, inventor and mechanical engineer who is best remembered now for originating the concept of a programmable computer.

CRITERIA
This award covers all aspects of parallel computing including computational aspects, novel applications, parallel algorithms, theory of parallel computation, parallel computing technologies, among others.

AWARD & PRESENTATION
A certificate and a $1,000 honorarium presented to a single recipient. The winner will be invited to present a paper and/or presentation at the annual IEEE-CS International Parallel and Distributed Processing Symposium (IPDPS).

NOMINATION SUBMISSION
Open to all. Nominations are being accepted electronically at www.computer.org/web/awards/charles-babbage. Three endorsements are required. The award shall be presented to a single recipient.

NOMINATION SITE
awards.computer.org

AWARDS HOMEPAGE
www.computer.org/awards

CONTACT US
awards@computer.org

IEEE CLOUD COMPUTING

May/June 2018
Vol. 5, No. 3

www.computer.org/cloud

TABLE OF CONTENTS

Cloud Reliability

Feature Articles

Columns and Departments

Also in This Issue

ISSN: 2325-6095 Copublished by the IEEE CS and IEEE ComSoc

Cloud Computing Reliability—Failure is an Option

Mazin Yousif
T-Systems, International

Cloud reliability is a measure of the probability that the cloud delivers the services it is designed for. This implies that the service is available, and performs in the way intended. When we use cloud services, it is easy to assume that they will deliver what they are designed and marketed to deliver. However, the cloud, just like everything else we develop, will fail at some time, due to hardware failures from natural disasters or human issues, unexpected software failures, or massive-scale cyberattacks such as Distributed Denial of Service attacks. It is not a matter of *if*, but rather *when*. It is also not a matter of a specific provider; they have all had outages.

When a Cloud Service Provider (CSP) delivers cloud services to millions of people and businesses, including businesses that rely on the cloud to deliver services to millions of customers, any failure could have major ramifications on the future of that CSP or business. There are many famous examples over the past few years, such as an AWS Elastic Load Balancer issue that impacted many customers, including Netflix, leading to the loss of service for tens of millions of Netflix customers on a Christmas Eve. There are many more examples of failures, some of which were documented in 2011 (www.infoworld.com/article/2622201/cloud-computing/the-10-worst-cloud-outages--and-what-we-can-learn-from-them-.html) and others that were documented in 2017 (www.computerworlduk.com/galleries/infrastructure/ten-datacentre-disasters-that-brought-firms-offline-3593580), including the worldwide British Airways failure. A major lesson is that most major failures are caused by humans. That is why CSPs augment their cloud infrastructure with technologies, architectures, resources, organization designs, processes, and workflows to make sure the infrastructure delivers its cloud services, is secure from attack, and to help protect against human error. Another potential source of failure is the interaction between applications and the cloud infrastructure. In a PaaS, for example, legacy software with its own optimizations for accessing local storage may conflict strongly with the CSP optimizations leading to performance degradation and possibly failure.

Examples of technologies and architectures include redundancies or tooling to monitor and provide visibility or Artificial Intelligence and analytics to predict failures or bottlenecks. Examples of resources and organization designs can be in the form of having teams responsible for manag-

Copublished by the IEEE CS and IEEE ComSoc
2325-6095/18/$33.00 USD ©2018 IEEE

ing backup and disaster recovery instead of single person, or conversely, only having a single person or role that has to approve changes. Examples of processes can be in the form of replication management, recovery management and fault masking. Examples of workflows can be in the form of triggered automated live migration when a certain load on a Virtual Machine (VM) or server or rack is reached. Companies have been trying innovative techniques beyond the traditional approaches we have used for years. For example, Netflix introduced the Chaos Monkey tool, which is designed solely to test how reliable operations are in the presence of failures. It kills random VMs or application modules to make sure that component outages will not disrupt the system as a whole. This approach has proved to be very effective, evolving to become a comprehensive suite of tools, referred to as the Simian Army, which is a collection of open source tools to test the resiliency of cloud operations covering reliability, security and recoverability. Examples of what is in this army are Latency Monkey, Doctor Monkey, Janitor Monkey, Security Monkey and Chaos Gorilla. I want to highlight two things about the Chaos Monkey approach and why it has proved effective: first, it creates disruption with an element of surprise and second, it proves that *the best way to avoid failure is to persistently fail.*

But no matter how good cloud reliability gets, application developers and other cloud customers have to design software architectures to tolerate cloud failures; including code design itself. We have been able to do because we have been good at identifying and understanding the majority of failures in our basic units of architectures. By basic units of architecture, I refer to units of deployments such as servers, switches, routers, firewalls, and storage appliances. Types of failures are numerous and can range from network or port failures to storage, to execution environments and many others. In other words, the key is to anticipate failures like these while the software architecture is designed.

Calculating the reliability of cloud services in a given hyper-scale datacenter with its many diverse heterogeneous efforts and features to boost reliability is a monumental task. But I am sure it can be done. I urge the community to investigate how best to determine reliability indices for such datacenters either formally or through simulations such as Agent-Based Modeling (ABM), where agents could represent the basic units of deployment mentioned earlier. In addition to the scale of the datacenter, the main crux of the challenge is the interaction among every hardware, software and mechanical component in the datacenter and the timescale of its operations. For example, we should be able to simulate how micro-failures, when they happen in mass and in any variety, propagate inside a datacenter and their overall impact on the datacenter as a whole.

Artificial Intelligence (AI) can also be very effective in boosting the reliability of cloud datacenters. We can collect data from every device and corner in the datacenter and if we place all that data in a datacenter datalake, we can analyze that data for all types of objectives including improving reliability, availability, efficiency, and security as well as reducing and optimizing energy consumption. Other emerging technologies, such as virtual, augmented and mixed reality, may also be of use. I also urge the community to investigate how best to leverage AI and other emerging technologies for the benefits of cloud datacenters and services. In fact, I am looking for authors to help here as I am editing a book on Intelligence in the Cloud and I am in the process of inviting authors to contribute to it. So if you think you can contribute, reach out to me.

This special issue on Cloud Reliability brings together an interesting collection of columns and papers. I, especially, urge our readers to check the Guest Editor Introduction because it provides a great overview of cloud reliability and also details the contents of the special issue.

About the Author

Mazin Yousif is the Editor-in-Chief of *IEEE Cloud Computing*. He's the chief technology officer and vice president of Digital Transformation at T-Systems International. He has a PhD in computer engineering from Pennsylvania State University. Contact him at mazin@computer.org.

What is “Cloud”? It is time to update the NIST definition?

Christine Miyachi
Xerox Corporation

IaaS, PaaS, and SaaS were formally defined in 2011. Have these definitions held up in the fast-moving world of Cloud Computing?

Back in 1985, I logged onto a Unix computer in Southern California as a guest user, while sitting in a lab in Cambridge, Mass. We all take such capabilities for granted now, but at the time it was magical. I was using ARPANET, a precursor to the Internet, and even though the only thing I could do was roam around directories using the command line, I was blown away. It was like I had teleported to a new world. ARPANET diagrams from that time (see Figure 1), show cloud-like structures and we sometimes used the word “cloud” to describe them.[1]

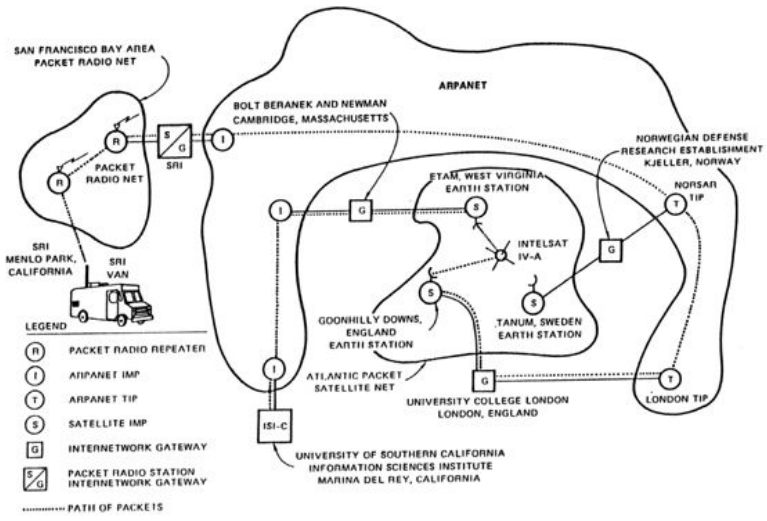


Figure 1. A diagram of ARPANET from the 1980s

Copublished by the IEEE CS and IEEE ComSoc
2325-6095/18/$33.00 USD ©2018 IEEE

Amazon Web Services (AWS) was one of the first companies to use the word "cloud" in their product advertising[2] in 2006 and then Cloud Computing showed up on the Gartner Hype Cycle and now it has its very own hype cycle.[3] And once the term was coined everybody started asking "What is cloud computing?"

DEFINITIONS

Enter the National Institute of Standards & Technology (NIST), a U.S. government entity that formally defines standards, metrics, and the like. After several years of work, industry collaboration, and multiple review cycles, they released the final version of the widely cited "The NIST Definition of Cloud Computing" in 2011.[4] In this publication, they define the now ubiquitous terms of SaaS, PaaS, and IaaS as follows:

- *"Software-as-a-Service (SaaS).* The capability provided to the consumer is to use the provider's applications running on a cloud infrastructure. The applications are accessible from various client devices through either a thin client interface, such as a web browser (e.g., web-based email), or a program interface. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user specific application configuration settings.
- *Platform-as- a-Service (PaaS).* The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment.
- *Infrastructure-as-a-Service (IaaS).* The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, and deployed applications; and possibly limited control of select networking components (e.g., host firewalls)."

Figure 2 shows these terms.

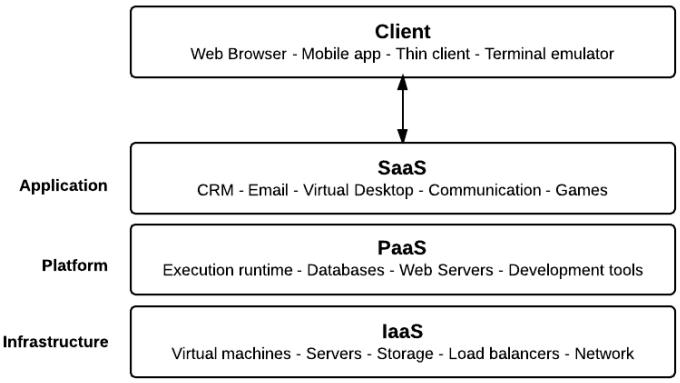


Figure 2. Basic Cloud Computing Definitions

IaaS, PaaS, and SaaS definitions have been remarkably resilient, and most cloud computing providers still use these terms on their marketing materials. But the definition is now seven years old and the industry has evolved rapidly to include diverse technologies such as containers and serverless computing. Does it need to be updated?

ANYTHING AS A SERVICE

One evolution of Iaas/PaaS/SaaS is Anything as a Service, typically referred to as XaaS. The definition of XaaS is "any technology delivered over the Internet that used to be delivered on-site."[5] Cloud providers are able to offer other technologies as cloud services now because Internet access has become increasingly reliable and faster, and server virtualization and serverless advances make powerful computing platforms and services readily available. XaaS allows for quick response to market changes. Some of the latest on XaaS is mentioned in the Gartner Hype Cycle[3] and elsewhere[6]:

- Blockchain PaaS[7]—Microsoft, AWS, and IBM are offering platforms that include tools for creating blockchain applications.
- Business Process as a Service (BPaaS)[8]—delivery of business process outsourcing services that are sourced from the cloud and constructed for multitenancy. Services are often automated, and where human process actors are required, there is no overtly dedicated labor pool per client. The pricing models are consumption-based or subscription-based. As a cloud service, the BPaaS model is accessed via Internet-based technologies.
- Database as a Service (DBaaS)[9]—a cloud computing service model that provides users with some form of access to a database without the need for setting up physical hardware, installing software or configuring for performance. All of the administrative tasks and maintenance are taken care of by the service provider so that all the user or application owner needs to do is use the database.
- Function as a Service (FaaS)[10]—a category of cloud computing services that provides a platform allowing customers to develop, run, and manage application functionalities without the complexity of building and maintaining the infrastructure typically associated with developing and launching an app.
- Malware as a Service[11]—a prosperous business run on the black market. Anyone can purchase code that will cause harm to computers or even hold it for ransom.
- Windows as a Service[12]—this doesn't fit the traditional definition of an XaaS. The user still has a version of the operating system (OS) running on their computer, but the OS management and updates are seamless to the user and managed in cloud. The OS is version-less in most respects because it is continuously being updated.

While most of these services fit in the traditional definitions, serverless computing is hard to define. There are no virtual machines to create and the PaaS vendor figures out the best way to run your functions. Where would you put serverless computing in the current model? And Windows as a Service turns the definitions on its head by make the client the OS instead of a traditional application. Clearly the definitions we have don't always apply.

A NEW MODEL

Johan den Haan, CTO at Medix, is an expert in model-driven engineering. He has created an interactive model and a more granular framework for Cloud Computing that may be a better fit to today's range of technologies and commercial providers offer.[13] His model builds upon the pre-existing NIST definition and related ones and integrates them into a layered framework. In addition, he partitions the PaaS and other layers into further sub-layers to align with the evolution of commercial services. Amazon, Microsoft, Google, and IBM all offer sophisticated PaaS solutions for Internet of Things, Big Data, and more. This model also brings in virtualization and software-defined control of networks and storage as base services, which had been missing. I've updated this model to include Serverless computing and some of the latest PaaS offerings we have in 2018. To me, serverless fits above PaaS while others have suggested it fits closer to IaaS. Kong Yang, head geek at SolarWinds said "Serverless architecture, aka functions as a service, will eliminate and replace IaaS for some use cases, especially where a function or series of functions can provide the needed application agility, scalability and flexibility without the application life-cycle management responsibility."[14]

6	SaaS		Applications		*End User*
5	App Services	Apps in the Cloud	Communications and Social Media Apps	Data as a Service	*Any User*
4	Built-up PaaS	Business as a Process	Social Media PaaS	Data Analytics	*Rapid Developers*
3.5		Serverless Computing			*Speed Developers*
3	PaaS				*Developers*
2	Foundational PaaS	Application Containers	Routing, Messaging, Orchestration	Object Storage	*DevOps*
1	Software Defined	Virtual Machines	Software Defined Networks (SDN)	Software Defined Storage	*Infrastructure Engineers*
0	Hardware	Servicers	Switches, Routers	Storage	
		Compute	***Communicate***	***Store***	

Figure 1. An updated model of cloud computing based on Johan den Hann's model

This model is helpful not only in organizing existing services, but also in visualizing future cloud computing growth areas. Johan den Hann says "I think the popular wisdom that cloud comes in three flavors (IaaS, PaaS, SaaS) is not providing a realistic picture of the current landscape. The lines between these categories are blurring and within these categories there are numerous sub-categories that describe a whole range of different approaches."[15] While Layer 2 is focused on PaaS for deployment of binaries, he says Layer 3 is for deploying code, and Layer 4 is specifically for a business engineer. Layer 2 is for DevOps where Layer 3 is for a professional developer. Layer 4 is for a non-professional developer that needs to connect applications together but doesn't actually write code. This layer contains Model-Driven Development which allows users to do their own programming via models. On the base layer of the model is Compute, Communicate, and Store—areas that have examples in all the vertical layers. IFTTT[16] (If This, Then That; a tool on the Web that allows you to connect different cloud applications and devices to each other) is an example of a Model-Driven PaaS tool in the Communicate column in Layer 4. This model clearly shows that there is more to the current cloud landscape then just SaaS, PaaS, and IaaS as originally conceptualized and documented in the "final" NIST framework.

GROWTH IN IAAS, PAAS, AND SAAS AND SERVERLESS

IaaS,[17] PaaS, and SaaS[18] are still growing at a rapid pace. Companies with aging infrastructures are replacing hardware with IaaS. And IT departments across the world are using SaaS to provide their employees with enterprise applications (such as email, storage, and word processing applications) as well as to provide customers with applications, such as package tracking for a logistics firm or catalog and shopping cart for an ecommerce firm. The number of offerings for PaaS continues to grow with companies like Amazon and Microsoft offering platforms specific to what applications and underlying infrastructure customers want.

Amazon, Google, and IBM also offer serverless computing options, also known as cloud functions. Serverless allows engineers to load code for only individual functions and have them run without worrying about machine or loading issues. For example, consider validating a credit card from a shopping cart application. In the past, you might spin up a virtual machine, load the credit card validation module, set up any network connections, call it, and then shut down the instance, perhaps being billed for an entire hour. With serverless, you just call the function and the cloud provider takes care of the rest, and you would be billed just for the resources used for a few milliseconds. If traditional enterprise computing was like buying a car, and traditional cloud computing was like renting it for the day, serverless is like taking a taxi.

CONCLUSION

In 1985, ARAPNET was about connecting computers and the original use of "clouds" to represent amorphous network constructs where the details weren't critically salient was born. As Cloud Computing emerged, the use of amorphous clouds to represent compute and storage as well as networking services was born, and NIST provided definitions for us to understand those services. Perhaps the authors at NIST will update their model as definitions from standards organizations are more widely vetted and accepted. For now, IaaS/PaaS/SaaS remained firmly perched as the best simple definitions and they are sticking because they are still largely accurate. Although newer models better portray what is happening in cloud, the simple explanation of SaaS, PaaS, and IaaS helps people get started in the complex world of cloud.

REFERENCES

1. "Internet History of 1970s," Computer History Museum; www.computerhistory.org/internethistory/1970s.
2. "Announcing Amazon Elastic Compute Cloud (Amazon EC2) - beta," Amazon; https://aws.amazon.com/about-aws/whats-new/2006/08/24/announcing-amazon-elastic-compute-cloud-amazon-ec2---beta/.
3. D.M. Smith and E. Anderson, "Hype Cycle for Cloud Computing, 2017," Gartner; www.gartner.com/doc/3772110/hype-cycle-cloud-computing-.
4. P. Mell and T. Grance, *SP 800-145, The NIST Definition of Cloud Computing*, NIST; https://csrc.nist.gov/publications/detail/sp/800-145/final.
5. D. Rahko, "The Realities of XaaS (Everything-as-a-Service)," *Bits & Bytes*, blog, 9 September 2016; www.modernmsp.com/realities-everything-service-xaas.
6. "What is XaaS (Anything-as-a-Service)?," *NH Learning Solutions Blog*, blog, 12 September 2017; https://blog.nhlearningsolutions.com/blog/what-is-xaas-anything-as-a-service.
7. D. Joshi, "IBM, Amazon & Microsoft are offering their blockchain technology as a service," *Business Insider*, 24 October 2017; http://www.businessinsider.com/ibm-azure-aws-blockchain-service-2017-10.
8. J. Den Haan, "The cloud landscape described, categorized, and compared," *The Enterprise Architect*; http://www.theenterprisearchitect.eu/blog/2013/10/12/the-cloud-landscape-described-categorized-and-compared.
9. "Database as a Service (DBaaS)," *Technopedia*; www.techopedia.com/definition/29431/database-as-a-service-dbaas.
10. "Function as a service," *Wikipedia*, 2018; https://en.wikipedia.org/wiki/Function_as_a_service.
11. M. Moreno, "Malware as a Service: As Easy As It Gets," *Webroot*, 31 May 2016; www.webroot.com/blog/2016/03/31/malware-service-easy-gets.
12. "Overview of Windows as a service," *Windows IT Pro Center*, Microsoft, 1 June 2018; https://docs.microsoft.com/en-us/windows/deployment/update/waas-overview.
13. "The cloud landscape described, categorized, and compared," *The Enterprise Architect*, blog; www.theenterprisearchitect.eu/blog/2013/10/12/the-cloud-landscape-described-categorized-and-compared.
14. C. Parizo, "The State of IaaS: Growing as Cloud Adoption Continues," *TechRepublic*; www.techrepublic.com/article/the-state-of-iaas-growing-as-cloud-adoption-continues.
15. "Categorizing and Comparing the Cloud Landscape," *The Enterprise Architect*; www.theenterprisearchitect.eu/blog/categorize-compare-cloud-vendors.
16. "IFTTT is a free platform that helps you do more with all your apps and devices," IFTTT; https://ifttt.com/about.
17. C. Parizo, "The State of IaaS: Growing as Cloud Adoption Continues," *TechRepublic*; www.techrepublic.com/article/the-state-of-iaas-growing-as-cloud-adoption-continues.
18. N. Gagliordi, "SaaS Leads Cloud Services Market Growth in 2017," *ZDNet*, 6 November 2017; www.zdnet.com/article/saas-leads-cloud-services-market-growth-in-2017.

ABOUT THE AUTHOR

Christine Miyachi is a systems engineer at Xerox Corporation and holds several patents. She works on Xerox's Extensible Interface Platform, which enables developers to create applications that work with Xerox devices by using standard web-based tools. Miyachi graduated from the University of Rochester with a BS in electrical engineering. She holds two MIT degrees: an MS in technology and policy/electrical engineering and computer science and an MS in System Design and Management. Contact her cmiyachi@alum.mit.edu.

The Next Grand Challenges

Integrating the Internet of Things and Data Science

Rajiv Ranjan
Newcastle University

Omer Rana
Cardiff University

Surya Nepal
Data61, CSIRO

Mazin Yousif
T-Systems, International

Philip James, Zhenyu Wen, Stuart Barr, Paul Watson
Newcastle University

Prem Prakash Jayaraman, Dimitrios Georgakopoulos
Swinburne University of Technology

Massimo Villari, Maria Fazio
University of Messina

Saurabh Garg
University of Tasmania

Rajkumar Buyya
University of Melbourne

Lizhe Wang
Chinese Academy of Sciences

Albert Y. Zomaya
University of Sydney

Schahram Dustdar
TU Wien

Editor: Rajiv Ranjan: raj.ranjan@ncl.ac.uk

This article discusses research challenges related to devising a new IoT programming paradigm for orchestrating IoT applications' composition and data processing across heterogeneous computing infrastructure (Cloud, Edge, and Things).

In the last decade, we have been transitioning from a data-poor to a data-rich world with the promise of unparalleled intelligence. Such transition will definitely require significant investments in every aspect in our societies including social, political, economic and cultural. Much of the (unprecedented) increase in data generation can be attributed to the abundance of mobile devices and wearables, the increase of instrumentation in every industry vertical, the mass adoption of social networks and the digitization of every aspect of our lives. Generically, the bulk of such data collection falls under the Internet of Things (IoT).[1–5] IoT data comes from a variety of sources that can be classified into (a) machine-based (e.g., environmental, weather, air quality, water quality, flows, traffic speeds, people flows and GPS location) or (b) people-based (e.g., social media, crowd sourced data collection, and simple text messaging) providing data and situational observations associated with events.

The emergence of computing paradigms such as Edge, Fog, and Osmotic Computing for supporting the analysis of data near the data sources are especially applicable for IoT use cases where insights need to be actioned on in the least amount of time possible.[1,6] Figure 1 depicts a typical IoT application infrastructure consisting of the *Things*, the *Edge*, and the *Cloud* layers. The layers are connected to each other in a plethora of ways. But the most interesting one is connecting the Things to the Edge of directly to the

Copublished by the IEEE CS and IEEE ComSoc
2325-6095/18/$33.00 USD ©2018 IEEE

cloud. Examples of networking protocols include (but not limited to) WiFi, Cellular (e.g., 4G & 5G), Bluetooth, Bluetooth Low Energy, LoRa-WAN [Lora], and Narrowband IoT (NB-IoT). On the other hand, the Edge layer consists of network gateways/middleboxes, Content Delivery Networks (CDNs), or micro datacenters, which provide limited computing and storage resources. The edge resources usually communicate with Cloud layer via wide Area Networks (WANs). The last layer is the Cloud, which is provided by different cloud providers such as Amazon, Microsoft, Tencent, Google and Alibaba. Cloud datacenters offer unlimited computational resources and their cloud services are usually offered on a pay-as-you-go fashion.

The increase in data collection, along with advances in infrastructure development and intelligence, have led to an opportunity for developing several new usage scenarios, ranging from smart cities, smart transportation, smart health care, to Industry 4.0 as depicted in Figure 2. However, the potential of these different paradigms/technologies requires coordination across several layers, leading to important research challenges to be addressed. Currently, existing IoT applications processing data run on remote Cloud infrastructure. To support new application scenarios, novel software/application abstractions are needed that can utilize distributed and dynamic infrastructure supported at Edge and Things layers (as shown in Figure 1). Moreover, IoT data is typified by the heterogeneity of data formats and types, which usually results in bespoke platforms and code that make subsequent integration and processing problematic and time-consuming. The provenance of data is another key aspect that IoT needs to address, not just to ensure the physical integrity of bytes produced, but to be able to trace decision making from model outputs to individual sensors or sensor platforms. This is significant to enable "trust" to be established in the analysis that is carried out on such data. IoT systems currently deployed are largely passive observers of the environment that transmit data to a remote location (with a varying and limited degree of on-board processing). Retasking this one-way behavior in a reliable fashion (e.g. changing sampling rates triggered by external stimuli) is a prerequisite for developing and deploying future IoT applications.

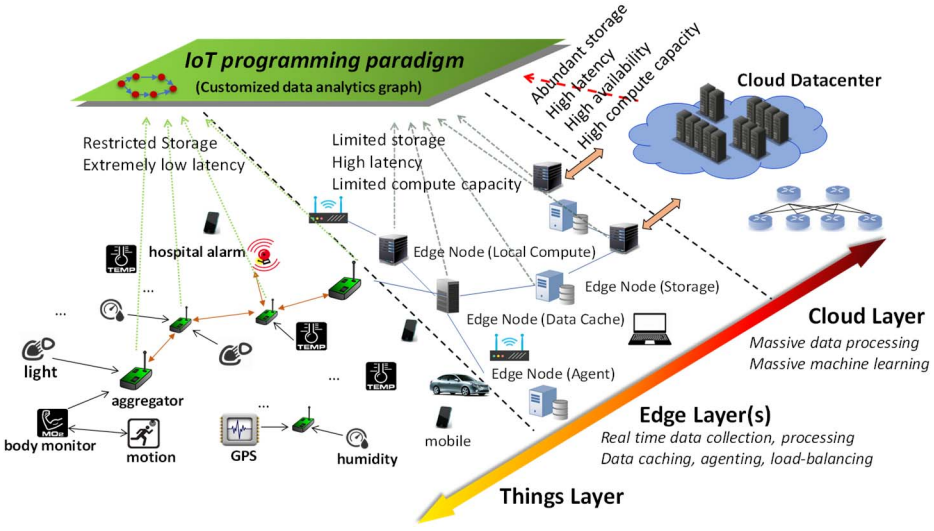


Figure 1. A typical IoT application infrastructure of a healthcare use case showing of Things, Edge, and Cloud layers.

In this column, we provide an IoT roadmap, moving from the (significant) existing research focus on handling streaming IoT data to developing application instances that have intelligence to adapt their behavior/operation based on several external (e.g. environment) and internal (e.g. application) QoS stimuli. We present our vision and associated challenges in the areas of IoT: (i) application composition, (ii) dynamic data management, and (iii) service orchestration across Things/Edge/Cloud infrastructure. Our discussion is based around the IoT application architecture shown in Figure 3 that illustrates how user requirements can be coupled with efficient utilization of computing resources and data provided by Cloud, Edge, and Things layers.

Figure 2. Examples of an IoT-driven smart world: from Smart Homes to Smart Retail, Industry 4.0 and Smart Grids.

In this environment, a user submits requirements to an IoT application orchestrator which identifies: types of services, data sources, QoS metrics that need to be monitored to meet user requirements. Following that, the orchestrator generates a graph showing services needed to realise application requirements. A data management component subsequently maintains and monitors IoT data sources including data governance, data analysis, and data warehousing. An IoT application orchestrator therefore considers IoT data sources as services with specific functions and Service Level Agreements (SLAs). The generated application graph will be deployed to computing resources according to the provisioned data resources and other SLA constraints. This deployment is not undertaken in a one-shot manner, requiring refinement and adaptation based on changes in the operating environment.

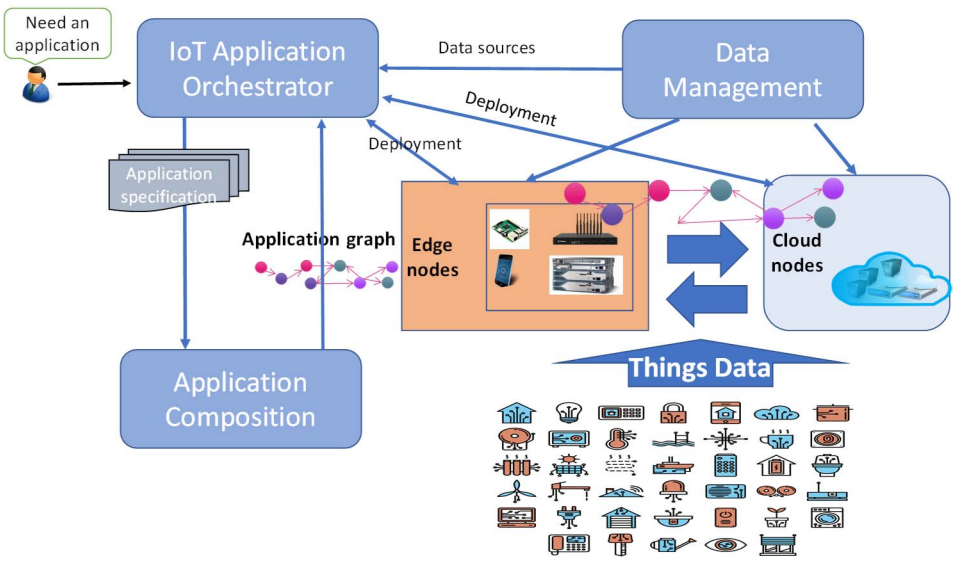


Figure 3. IoT application orchestration, showing interplay among different system components.

IOT APPLICATIONS COMPOSITION

IoT Data Sources

In general, IoT data can be of different types and can be collected at different rates and time scales. The data can be generated by two types of sources: Things (e.g. environment monitoring sensors, GPS, etc.) and People (e.g. social networking apps). Things, such as sensors, provide quantitative observational values; they provide measurement of physical phenomenon at different levels of precision. Moreover, these measurement data can be generated in different formats such as images from cameras, audio from satellites, and text from GPS. Conversely, social sensors provide a qualitative observation of a situation very quickly and succinctly. An IoT application, such as real-time flood forecasting and warning, requires the integration of machine and social sensors data to provide complementary and corroborative information. This aggregate data can be semantically tagged to generate and distribute events of interest (to particular subscribers). One of the key data science research question is how to identify IoT data sources that are most appropriate for a given IoT application context/use case. To answer this question, we need to overcome the following five challenges summarized by Baltrusaitis and colleagues:[7]

1. **Representation:** Structure and represent the data to facilitate multiple modalities, exploiting the complementarity and redundancy of different data sources.
2. **Translation:** Interpret data from one modality to another, i.e., provide a translator that allows the modalities to interact with each other for enabling data exchange.
3. **Alignment:** Identify the relation among modalities. This requires identifying links between different types of data.
4. **Fusion:** Fuse information from different modalities (e.g., to predict).
5. **Co-learning:** Transfer knowledge among modalities. This explores the field of how the knowledge of a modality can help or enhance a computational model trained on a different modality.

A Standard way to describe an IoT Computation Unit

There is a requirement to define a basic IoT computation unit (a software abstraction) that can be ubiquitously deployed across different infrastructures (as proposed by the Osmotic Computing programming paradigm)[1] and can be migrated based on various potential "triggers (e.g., performance, security/privacy or cost)". A software abstraction is used to describe a basic IoT computation unit called MicroELement (MEL).[2] A MEL encapsulates:

1. **MicroServices (MS),** which implement specific functionalities and can be deployed and migrated across different virtualized and/or containerized infrastructures (e.g., Docker) available across Cloud, Edge, and Things layers;
2. **MicroData (MD),** encodes the contextual information about (i) the sensors, actuators, edge devices, and cloud resources it needs to collect data from or send data to, (ii) the specific type of data (e.g., temperature, vibration, pollution, pH, humidity) it needs to process, and (iii) other data manipulation operations such as where to store data, where to forward data, and where to store results;
3. **MicroComputing (MC),** executing specific types of computational tasks (machine learning, aggregation, statistical analysis, error checking, and format translation) based on a mix of historic and real-time MD data in heterogeneous formats. These MCs could be realized using a variety of data storage and analytics programming models (SQL, NoSQL, stream processing, batch processing, etc.); and
4. **MicroActuator (MA),** implementing programming (e.g., for sending commands) interfaces with actuator devices for changing or controlling object states in the IoT environment.

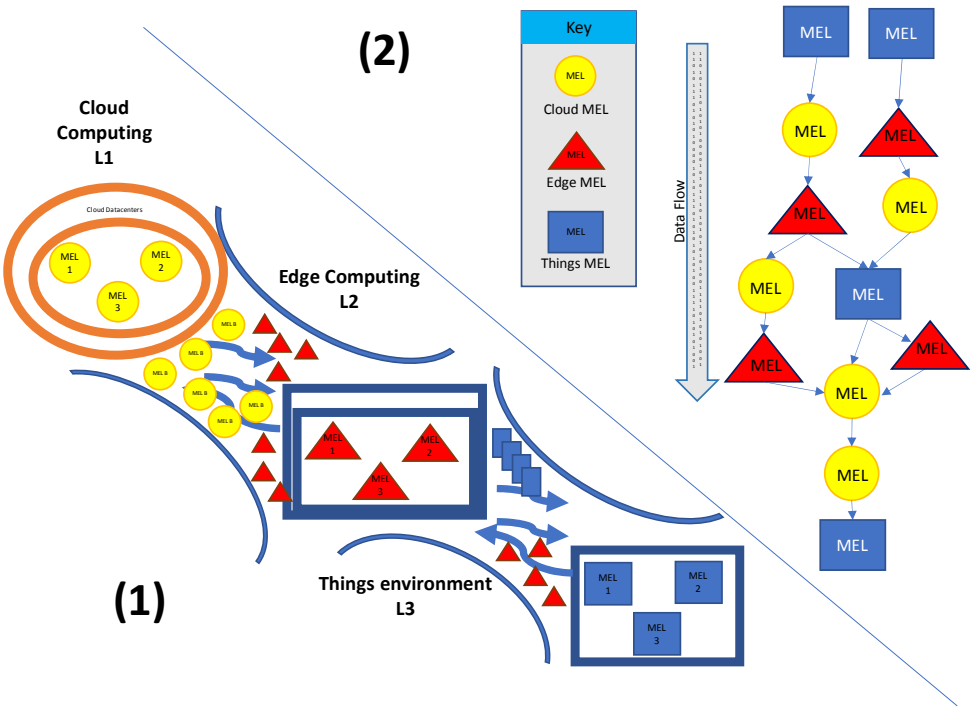


Figure 4. Osmotic "movement" of (1) MELs across Cloud, Edge, and Things; (2) MELs graph representation.

In summary, developments such as MEL,[2] provide a good starting point for describing the basic IoT computation unit. However, additional enhancements to the MEL abstraction are required so that it can be used to describe and program different types of IoT applications (e.g., Smart Homes, Smart Grids). Current IoT applications development is typically a vertical, proprietary application stack that is often difficult to generalise. Where there is heterogeneity of IoT sensors and platforms (Smart Cities) they are typified by large amount of bespoke code and data integration requirements. Adoption of standards and protocols across IoT deployments is piecemeal at best and chaotic at worst. A step-by-step development approach reducing the cost of deployment and configuration, putting flexibility of system design at the core, could be used to increase uptake of software abstractions such as MEL.

IoT Application Graph Choreography

We need fundamentally new IoT applications programming pattern (e.g., MEL graph as shown in Figure 4) for: (1) decomposing IoT data analysis activities into fine-grained activities (e.g., statistics, clustering, classification, anomaly detection, accumulation, filtering), each of which may impose different planning and run-time orchestration requirements; (2) identifying and integrating real-time data from IoT devices and historical IoT data distributed across Cloud and Edge resources; (3) identifying data and control flow dependencies between data analysis activities focusing on coordination and data flow variables, as well as the handling of dynamic system updates and re-configuration; and (4) defining and tagging each data analysis activity with run-time deployment constraints (QoS, security and privacy).

Existing composition and choreography standards (such as SOAP, TOSCA, and BPEL) are not suitable, as they *cannot sufficiently* capture the complexity of an IoT application graph (e.g., heterogeneous data sources, data and control flow dependencies across heterogeneous activities, and heterogeneous software and hardware configurations across Things, Edge, and Cloud layers). Additional research is required for implementing and deploying an IoT application graph (see Figure 4). Approaches such as Juju and Fabric8 provide promising developments in this area.

IoT Application Graph Performance Calibration

IoT applications developers need to systematically undertake performance characterization of data analysis activities (e.g., MELs) across different parts of the infrastructure (Cloud, Edge, and Things). They need to understand and reason about most important QoS metrics and/or security and privacy threats to each data analysis activity. For instance, QoS metrics required to characterize the performance of a data analysis activity mapped to a Cloud layer (see Figures 1 and 3) is quite different from a gateway and/or device in the Edge and Things layer Similarly, performance analysis[8] at the Things and Edge layers may require assessing network stability, throughput optimality, routing delays, fairness in resource sharing, available bandwidth, and sensor battery state. These QoS metrics might be very different from the ones relevant to a Cloud operator, who is interested in end-to-end response times, platform scalability and reliability, virtual server utilizations, and the costs of moving data to and from the Cloud.

Currently, many benchmarking kernels (e.g., TPCx-IoT, BigDataBench, TeraGen, TeraSort, TeraValidate, Google ROADDEF, Linear Road, DeepBench, and MLPack) exist for characterizing performance of IoT data analysis activities. Moreover, each benchmarking kernel type has its own benefits and can help us understand performance of specific type of IoT data processing activity under variable workload scenarios. For example, TPCx-IoT benchmark can be used to calibrate the performance at the Edge layer as it is representative of the data analysis activities (data aggregation, real-time analytics and persistent storage) that are typically hosted in IoT gateway systems. Similarly, Google ROADEF & Linear Road benchmarking kernels can be applied for calibrating performance of stream processing data analysis activity at the Edge layer. On the other hand, TeraGen, TeraSort, and TeraValidate benchmarking kernels can be applied to calibrate the performance of batch processing activity at the Cloud layer. As it can be inferred, none, by themselves, can reveal the true bottleneck of whole IoT application graph, which includes multiple data analysis activities unless multiple benchmarking kernels are properly combined together. Hence, one of the possible research directions will be to identify/build different suitable benchmarks from each type of the data analysis activities and hierarchically/logically combine them to draw accurate conclusions across an IoT graph in a holistic way.

IOT APPLICATION DATA MANAGEMENT

The ability to efficiently capture and manage multiple, diverse types of real-time and historical data streams lie at the heart of developing improved decision models and impact analytics. This includes traditional, structured data such as that acquired by environmental sensor networks, for instance. It also includes more challenging unstructured data streams including geospatial social media feeds (twitter, Instagram, news feeds, etc.), as well as data from the continuous monitoring of ambient environment, people, and machines. In addition to being harder to interpret and use, such data feeds have variable velocities and less structure and thus will require more opportunistic data management approaches.

Storage

It is now widely recognized that in the era of high velocity, volume and variety data no single data storage approach is optimal for IoT data management purposes. Thus, there is increasingly a move towards developing heterogeneous storage platforms where different data storage and analytics programming model (e.g., stream processing, batch processing, SQL, NoSQL) can be used depending on the subsequent analytics requirements. Hence, future research will need to address the challenges of selecting and tuning a suite of data storage and analytics programing models and related tools. To achieve this, it is necessary to take into account the specific characteristics of each IoT data stream, as well as the specific access requirements of the underlying IoT application. There is also growing interest in supporting storage at the network edge, eliminating the need to capture all data into a central repository, such as work proposed by Edge Analytics (www.edgeintelligence.com).

Access

The real-time and/or semi real-time data analysis requirements of IoT applications require seamless access to IoT data feeds. The research community will need to investigate techniques for efficient IoT data retrieval that will include indexing to meet the demands of real-time analytics and support for *sharding* of the data stream based on application and infrastructure requirements. Apache Kafka already provides some of this capability, i.e., the ability to shard a data stream based on the memory capacity of the computational nodes undertaking the analysis. However, understanding mechanisms to support such sharding at the network edge remains a challenge. One possible research direction will be to implement a federated approach; where data query and retrieval functionality from multiple individual platforms are mediated by an IoT programming abstraction (e.g., a MEL). As discussed earlier, MEL will need to expose a uniform programmatic interface (APIs) to models and analytics, hence, reducing the barriers (and associated latencies) to data ingestion into models and visualisations. The new federated API suite may utilize the Apache Spark SQL API to benefit from its existing interoperability features, in addition to other Apache libraries such as Samza and Kafka. The chosen API must be lightweight, and enable integration with services made available in libraries supported by other vendors.

Geo-Distributed Cross-Querying

As noted above, an IoT application may be described as a graph and stored across different parts of the infrastructure which are likely to be geo-distributed. The data sets that need to be processed through such applications are also distributed, requiring support for distributed search and for performing multiple analytical queries in a geo-distributed manner. Moreover, these analytical queries will differ depending on the type of storage and analytics programming model implemented by a given IoT data analysis activity. The analytical queries can be a SQL query, stream processing query, NoSQL query, or a MapReduce query. Hence the challenge is to design new types of *multi-query planning and provisioning algorithms* that can optimally distribute queries across data analysis activities mapped to different parts of an IoT infrastructure, while optimizing end-to-end QoS associated with the query graph (query plan), to improve resource utility and meet users' SLAs.

Existing geo-distributed querying systems[9–11] do not consider the heterogeneous computing infrastructure, neither do they execute the queries over different types data analysis activities programmed using heterogeneous models (e.g., stream processing, NoSQL, SQL, batch processing). While IoT applications need to process and query both static and real-time data, existing geo-distributed querying systems were designed for managing static data. Event Processing Language (EPL) has been used in majority of stream process platform such as Apache Spark, Kafka, Flink and Esper. Similar to SQL, EPL supports the following data querying operations: SELECT, FROM, WHERE, GROUP BY HAVING and ORDER BY. Unlike relational database systems, these EPL-based platforms can limit the query data size to guarantee real-time processing. However, one of the core limitations of EPL- and SQL-based querying approaches is that they cannot deal with heterogeneous data stored across multiple types of storage platforms and/or programmed using multiple types of storage and analytics programming models.

Integration

Future research efforts need to develop innovative techniques for supporting the integration of heterogeneous IoT data from thousands or millions of sources. However, establishing relationships between IoT data sources (e.g., CCTV), associated events (e.g., air pollution, traffic incidents, flooding, landslide), and stakeholders (e.g. decision makers, first responders) are generally difficult to detect as it is dependent on the context of the IoT application. For example, data integration techniques for air quality monitoring need to establish a relationship between road traffic patterns and the associated air and noise pollution. At the same time it should allow seamless integration of different types of air quality data including the pollution data extracted from the raw chemical sensors data, water quality sensors, traffic flow sensors (e.g., CCTV), and air quality sensors.

Moreover, such data integration efforts can build on existing standards such as the Semantic Sensor Networks (SSN) to allow consistent representation of IoT sensors and their data streams. Another very interesting research direction will be to apply graph-based approaches and machine learning techniques for discovering relationships between IoT data sources, associated events, and stakeholders.

Data Provenance

An IoT data provenance technique is responsible for logging the origins (IoT data sources) and historical derivations of data by means of recording data analysis transformation operations (those data analysis activities that are in charge of manipulating data). IoT-based sensing equipment is deployment to improve decision making. Automated decision making at the source (e.g., traffic control signal) requires metadata to be able to validate decisions post-event and check the reliability/efficiency of the decision-making processes. Once one steps away from real-time automated decision making, the process chain of data manipulation becomes more complex, and likewise, the associated decision making process moves further from the data source. Thus, the provenance and metadata of IoT systems are critical to the implementation, trust, and social use necessary for the deployment and use of IoT based sensing.

Dealing with provenance in the context of large IoT application graphs is challenging often due to the size and volume of data involved, the heterogeneous configurations of the underlying infrastructure (Cloud vs. Edge vs Things), and the heterogeneous data analysis activities (e.g., type of storage and analytics programming model). Developing contextual metadata is essential for reasoning about heterogeneous IoT data and related lifecycle activities (e.g., produce, store, process, and query). Traditional data provenance techniques require collection and transmission of large data volumes, which is impractical for IoT applications that warrant sub-second decision making and data processing latency. Hence, new techniques are required which can reduce and enhance the efficiency of provenance and metadata collection, recording, and transmission. Understanding how provenance relationships can be derived from IoT data processing activities therefore remains a challenge, as precedence relationships identifying which output was a consequence of a particular set of inputs may be difficult to establish.

One possible research direction to develop IoT data provenance technique based on Blockchain's Distributed Ledger Technology (DLT) to record lifecycle activities on data as it travels through the IoT ecosystem. Another hard challenge to solve will be to develop provenance techniques than can verify complying with data privacy regulation such as GDPR. Here GDPR-based IoT data access policies need to be verified, to ensure that the user has provided consent on how their data can be analysed and fused with other data sources. Undertaking GDPR compliance for statically held data (e.g. user information) can be easier to manage, however extending this to a dynamic data stream (which may be context dependent) remains a challenge. Another research topic is the use of smart contracts in IoT deployments that involve more than one vendor and where data exchange needs to take place. This will be needed in complex use cases such as smart cities.

IOT APPLICATION ORCHESTRATION

When an IoT application is expressed as a collection of multiple self-contained data analysis activities (e.g., MEL), future research will need to consider the following: (i) choosing storage and analytics programming models (e.g., stream processing, batch processing, NoSQL) and computational (e.g., data analysis algorithms) models that can seamlessly execute in highly distributed and heterogeneous IoT infrastructure (see Figure 3 and Figure 4); (ii) dynamically detecting faults across multiple parts of the IoT infrastructure; (iii) dynamically managing *resources, data, and software* available in Things, Edge and Cloud layers driven by IoT-specific applications requirements (data volume, data velocity, QoS, security, and privacy).

Optimal Configuration selection

Mapping of IoT application (graph of data analysis activities) demands selecting bespoke configurations[1] of resources at Things, Edge, and/or Cloud layers from abundance of possibilities. For example, in context of: (i) Cloud: we need to consider configurations such as datacentre location, pricing policies, compute/storage configurations, virtualization features, upstream/downstream network latency, etc. (ii) Edge: we need to consider configurations such as Edge device (Raspberry Pi 3, UDOO board, ESP8266) hardware features (e.g., CPU power, main memory size, storage size), upstream/downstream network latency, supported virtualization features, etc.; and (iii) Things: we need to consider data source location, battery, upstream/downstream network latency, network type, life, sensor type etc.. The diverse configuration space coupled with conflicting (trade-off) QoS, security, and privacy requirements leads to exponential growth of potential search space. Hence, computing a near-optimal solution for mapping IoT application graph to Things, Edge and/or Cloud layers in a reasonable time is NP-hard in much stronger sense when compared against task mapping and scheduling problems in Cloud computing, Services computing, and Grid computing systems.

Given the complexity of multilayered configuration search space and mix of conflicting requirements, future research efforts need to focus on developing computationally tractable optimisation techniques that can accommodate cross-layer resource configurations and conflicting QoS, security and privacy requirements. Moreover, these techniques will need to cater for diverse requirements of heterogeneous IoT applications.

Holistic Monitoring

To automatically predict and detect anomalies and their root causes, it is critical to monitor[12] and profile the following contextual information in real-time: QoS parameters (whole IoT application graph, activity-specific, edge-specific, Things-specific and cloud-specific) and activity-specific data flow. Much of the difficulty in monitoring IoT application graphs is due to the massive scale and heterogeneity of underlying computing infrastructure and multi-modal data sources.

Although QoS monitoring topic has attracted a lot of attention from the distributed computing (Grid, Cloud, Web Service) community, none of the existing monitoring tools and techniques are able to monitor performance of IoT application graphs in a holistic way. Data streams themselves need monitoring and here context and location can be important. Understanding a fault or change in a sensor may require profiling of the normal operation of that sensor which cannot be assumed to operate in a generic fashion. Hence, novel monitoring techniques providing detailed data flow and QoS information related to IoT application graph are required. These techniques will need to give deep insights into how data analysis tasks and underlying resources are performing, where possible QoS bottlenecks should be monitored along with all security or privacy threats. At the same time, these techniques should be able to give holistic view of QoS and data flow in an end-to-end fashion.

Fault-detection and Debugging

In future IoT environments where multiple decentralized and distributed devices and resources from the Things (e.g., sensors), Edge (e.g., compute and storage) and Cloud layers function together, the probability of failures will be high—simply due to the plethora of connected things. Moreover, the complexity of multiple data analysis activities simultaneously happening across these devices further adds to the chances of failure particularly in the cases when they are interdependent of each other. Such failures can be in several forms for example hardware, software or wrong user inputs or interaction with the ecosystem including connectivity, mobility and battery power of different devices. For example, some devices may be connected via wireless connections that may also vary in speed and reliability, this will impact the data transfer rate required by the applications. Most of the edge devices may have wireless connectivity and many cases they may be mobile devices, which are battery powered. In other words, their capacity and capability may vary quite frequently and some may fail if overloaded. In summary, the complexity of failure management within such heterogeneous and changing environments is not trivial. This gets

complicated when various data analysis activities have some SLA defining bounds on QoS parameters with IoT providers. Therefore, the challenge for IoT providers is how to ensure SLA /QoS in such failure-prone environments. Or question may be asked how to define QoS parameters in SLAs for such environments.

Run-time Reconfiguration

One of the most important aims of IoT applications orchestration processes is to design run-time reconfiguration algorithms to dynamically allocate and reallocate data analysis activities (within an application graph) to different parts of the infrastructure depending upon many unpredictable events including sudden unavailability of sensing devices (e.g., due to battery drain, a power failure, or IoT devices being made unavailable by their owners) and degradation of either an edge node or the communication network (e.g., due to overloading, edge failure or changes in the IoT data flow rate).To determine how each data analysis activity consistently achieves its QoS objectives while dynamically handling the run-time uncertainties of data flow behaviour and "Cloud-Edge-Things" performance is a unsolved research challenge. Moreover, data analysis activities are interdependent; changes in the execution and data flow of one activity will influence others. At run time, the reconfiguration technique must therefore be aware of these interdependencies between activities – passing aggregate data from the edge to the cloud, for example. In other words, all of the above uncertainties in IoT applications demand bespoke run-time reconfigurations.

To achieve multilayer ("Cloud-Edge-Things") reconfiguration, IoT research community needs to investigate a comprehensive set of QoS prediction models for heterogeneous data analysis activities mapped across Cloud, Edge, and Things layers. The QoS prediction models should be dynamically tuneable based on real-time monitoring information available from holistic monitoring approaches. These QoS prediction models will also need to undertake trade-off analysis between limited compute capability and low latencies at the Edge (i.e., close to the IoT devices), versus large compute capability and high latency (i.e., at the Cloud). In order to achieve this, new research efforts are required focussed on designing network, compute and storage aware optimisation algorithms to identify the best topology for IoT graph dataflow that may arise from the same underlying physical configuration of the edge and IoT networks.

CROSS CUTTING CONCERNS

Security, Privacy and Compliance

With increasing up take of IoT application services (smart city, smart traffic, smart home, smart healthcare), often hosted over (distributed cloud, edge, and IoT) infrastructure, there is a realization that IoT services can involve an interlinked set of providers (data, service, network and infrastructure). Stakeholders in IoT environments implicitly expect and demand their data and services to be secure, trusted as well as to preserve their privacy. Users of IoT applications may only interact with other applications via simple web interface without actually being aware of the large, distributed service, data, and network ecosystem. They often entrust their data and identity without realising that IoT applications providers may share their data with several back-end services (Cloud hosted analytics, mobile edge network provider, government stakeholders).

Security in IoT applications involves satisfying mainly two key properties: Authentication and Integrity. Achieving successful authentication in IoT ecosystem requires a device identity management and a suitable authentication scheme. The ubiquitous (and heterogeneous) nature of the variety of IoT devices from different vendors and their presence in an untrusted environment with no central authority makes the traditional enterprise-based identity management system incapable of working for IoT applications. The challenge for a further research is how to manage the identity of IoT devices in fully decentralised and distributed systems. A comprehensive identity management framework that works seamlessly with existing enterprise-based identity management systems is needed.

Traditional authentication schemes are no longer applicable to IoT environments due to limited resources (e.g., memory, battery life, etc.). Different lightweight authentication protocols have been explored in the literature to address this challenge. A practical lightweight authentication scheme still remains as a problem to be solved. In addition, many schemes under development have largely ignored the possible realisation of Quantum Computing. Developing lightweight post-quantum authentication schemes is the next grand challenge in IoT authentication. It is important to note that such authentication schemes should not only support devices to gateways/edges authentication, but also mutual authentication between devices and all other components in the system (e.g., other devices, servers, users, etc.).

Integrity is the most important security property when you consider the integration of IoT with the emerging data science paradigm. First, a device needs to be trusted so that the data generated by the device can be reliably used in making the (right) decision. Because of the diversity of devices and manufactures, many different firmwares could be present at any one point in time; all could be vulnerable to attacks by adversaries. Performing static and dynamic analysis might help to identify potential vulnerabilities, but it is almost impossible to do so for every firmware in the market. Furthermore, even as vulnerabilities are detected and a patch is developed to fix them, the distribution of such patches to all IoT devices is very difficult since the devices may or may not support automatic discovery. Further research is needed in this area. Second, we need to ensure that data integrity is maintained, not only when the data is at rest and in motion, but also while performing the data analytics. Providing integrity to data analytics is a challenging area of research, and a good amount of attention has been paid to it in recent time. Adversarial machine learning (e.g., GAN-based approaches) is an active research area and it is important to understand the impact of it on the integration of IoT and data science.

Privacy has been widely studied under different research disciplines: Privacy Preserving Technologies (PPT), Privacy Enhancing Technologies (PET) and Privacy Engineering (PE). While this has been a well-defined problem in the community, it is greatly exacerbated by the expansion of Internet-connected devices. A large number of techniques have been developed; most notably, ones in recent time include differential privacy and privacy-by-design. Some of these privacy technologies have been extended and used in IoT applications. This area needs further research in terms of developing privacy guidelines for IoT devices and data. General Data Protection Regulation (GDPR) introduced by the European Union (EU), which ensures that non-expert users can make informed decisions about their privacy and thereby give ‘informed consent’ to the use, sharing and repurposing of their personal data, is a right direction towards addressing some of the problems. Other world geographies are likely contemplating regulations similar to GDPR. Privacy preserving data trading platform is needed to ensure that IoT data can be made available for data science without worrying too much about privacy breaches.

Like all other computer systems, the weakest link in security and privacy is always the end users. Hence, the development and deployment of technological solutions should put users at the core. Human-centric security and privacy for IoT is the next big research challenge. Comprehensive security and privacy guidelines need to be developed for users, whether they are employees or citizens. Different levels of governments have a role to play in this space. Guidelines and regulations can only work if there is a way to check the compliance against such regulations and guidelines. A number of governments around the world have started to look at this seriously. The research challenge is how to automate the compliance checking process.

Ultimately, the IoT research community will need to investigate new security-, privacy- and compliance-aware applications graph provisioning mechanism to enable: (i) greater trust among users, stakeholders and IoT applications’ providers; (ii) emergence of new actors that can offer services; and (iii) an IoT data marketplace that enables greater control of personal data by users.

As IoT is directly involved in the physical world we are living in and the data we generate, security, privacy and compliance will always remain sensitive topics and must be managed carefully. That said we feel the aforementioned points are just snippets of what needs to be covered. So, we urge the research community and industry at large to considerably elaborate on these topics going forward.

Scalable and Unified messaging

An IoT application may use a hybrid approach for message communication depending on context (i.e., centralized and decentralized). The message communication includes sensor to sensor (S2S), sensor to edge (S2E), edge to cloud (E2C) and sensor to cloud (S2C) interaction. The use of centralized modes such as S2C cannot meet the requirements of soft and hard real-time applications. For example, a neighbouring smart vehicle system in New York city uses local WiFi to support the real-time interaction of vehicle to vehicle (V2V) and vehicle to traffic infrastructure (V2I).[13] Existing cloud-based communication solutions, such as AWS IoT and Azure IoT Hub, are unable to meet the strict QoS, security, and compliance requirements of diverse IoT applications. As we note in a previous Blue Skies instalment,[14] future efforts needs to focus on developing distributed IoT messaging middleware which can leverage the ever- increasing amount of resources at the edge of the network to provide reliable, ultra-low-latency, and privacy-aware message routing and communication. Having said that, the protocol heterogeneity inherent to Edge (e.g. WiFi, 4G/5G, wired) and Things (e.g. Bluetooth Low energy, COAP, MQTT) resources, and the unpredictability and resource constrained nature of Edge and Things resources, make it extremely challenging to provide resilient coordination mechanisms and guaranteed message delivery. In order to fit these heterogeneous protocols on the existing architecture such as OSI Model, we need to *unify and abstract* the protocols present across different layers into a *new IoT communication API stack*. The new IoT communication API will need to include communication adapters for multiple, heterogeneous communication protocols relevant to Things, Edge and Cloud layers.

Programmable networks for supporting IoT

The ability to independently manage the control and data plane, as proposed by software-defined networking (SDN) is an approach that allows network administrators to program and initialise, control, change and manage networking components of the OSI model. SDN is designed to address programmability shortcoming of static architecture of traditional networks such as those are used in current datacenters.[15] SDN has already shown great performance improvements in other fields such as flow optimisation or bandwidth allocation in cloud-based datacentres, and as yet has never been realistically utilised for IoT application infrastructure (see Figures 1–4). This is due to the fact that current SDN platforms are built on two assumptions: (1) having a centralized controller and (2) the requirement to contact devices to pull usage statistics or to push commands to devices. None of the above assumptions are applicable to IoT applications infrastructure because (1) connecting millions of IoT devices to a centralised controller is not scalable; and (2) IoT sensors/actuators may have intermittent connections. Thus, one of the important research direction will be to further study and consequently modify current SDN controllers (such as OpenDayLight) to first subdivide the controlling layer and secondly to tolerate lossy connections.

CONCLUSION

There is significant potential for IoT applications to improve our well-being and be drivers for social good. To go beyond the hype and the use of bespoke solutions, we need to address the many architectural challenges that will enable demonstrable and ongoing value. While many challenges exist firmly in the ICT sphere, these also accompany external factors and immature technologies elsewhere, so development must go hand-in-hand with e.g., more reliable and accurate sensing systems (and acknowledgement of the uncertainty in many systems we measure), better and more reliable communication stacks and improved power management and battery life. IoT applications also do not exist in a technology vacuum. They are scaffolded by existing regulatory systems, processes, social, economic and legal systems, which means holistic change may be required to achieve the IoT vision we have been promising the world. IoT infrastructure needs to be developed within a symbiotic feedback loop with these existing systems to engender public trust and the social and political license to benefit fully from technological advances.

It now also appears that Artificial Intelligence (AI) based technologies are the new operating system, as many hardware & software vendors attempt to integrate these in their IoT systems and software libraries. Understanding how these AI algorithms process and interpret such data remains a challenge, as opening up the "box" to understand the actual operations of these algorithms remains unclear, leading to issues around trust in how such algorithms make their decisions (but this is a topic for another column).

References

1. M. Villari et al., "Osmotic Computing: A New Paradigm for Edge/Cloud Integration," *IEEE Cloud Computing*, vol. 3, no. 6, 2016, pp. 76–83.
2. M. Villari et al., "Software Defined Membrane: Policy-Driven Edge and Internet of Things Security," *IEEE Cloud Computing*, vol. 4, no. 4, 2017, pp. 92–99.
3. L. Wang and R. Ranjan, "Processing Distributed Internet of Things Data in Clouds," *IEEE Cloud Computing*, vol. 2, no. 1, 2015, pp. 76–80.
4. H.L. Truong and S. Dustdar, "Principles for Engineering IoT Cloud Systems," *IEEE Cloud Computing*, vol. 2, no. 2, 2015, pp. 68–76.
5. A. Taivalsaari and T. Mikkonen, "A Roadmap to the Programmable World: Software Challenges in the IoT Era," *IEEE Software*, vol. 34, no. 1, 2017, pp. 72–80.
6. A. Dastjerdi and R. Buyya, "Fog Computing: Helping the Internet of Things Realize its Potential,," *Computer*, vol. 49, no. 8, 2016, pp. 40–4.
7. T. Baltrusaitis, C. Ahuja, and L.P. Morency, "Multimodal machine learning: A survey and taxonomy," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. Early access, 2018; doi.org/10.1109/TPAMI.2018.2798607.
8. G. Kecskemeti et al., "Modelling and Simulation Challenges in Internet of Things," *IEEE Cloud Computing*, vol. 4, no. 1, 2017, pp. 62–69.
9. R. Viswanathan, G. Ananthanarayanan, and A. Akella, "Clarinet: WAN-Aware Optimization for Analytics Queries," *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation* (OSDI 16), 2016.
10. Q. Pu et al., "Low Latency Geo-distributed Data Analytics," *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (SIGCOMM 15), 2015, pp. 421–434.
11. K. Kloudas et al., "Pixida: optimizing data parallel jobs in wide-area data analytics," *Proceedings of the VLDB Endowment*, vol. 9, no. 2, 2015, pp. 72–83.
12. A. Souza et al., "Osmotic Monitoring of Microservices between the Edge and Cloud," *20th IEEE International Conference on High Performance Computing and Communications* (HPCC 18), 2018.
13. k.m.d. Dikaiakos et al., "Location-aware services over vehicular ad-hoc networks using car-to-car communication," *EEE Journal on Selected Areas in Communications*, vol. 25, no. 8, 2007, pp. 1590–1602.
14. T. Rausch, S. Dustdar, and R. Ranjan, "Osmotic Message-Oriented Middleware for the Internet of Things," *IEEE Cloud Computing*, vol. 5, no. 2, 2018, pp. 17–25.
15. D. Cho et al., "Real-Time Virtual Network Function (VNF) Migration toward Low Network Latency in Cloud Environments," *Proceedings of the IEEE 10th International Conference on Cloud Computing* (CLOUD 17), 2017, pp. 798–801.

ABOUT THE AUTHORS

Rajiv Ranjan is a Chair Professor of Computing Science and IoT at Newcastle University, UK. He received a PhD in Computer Science and Software Engineering from the University of Melbourne. His research interests include Internet of Things, Big Data Analytics. Contact him at raj.ranjan@ncl.ac.uk.

Omer Rana is a full professor of performance engineering in the School of Computer Science and Informatics at Cardiff University. He received a PhD from the Imperial College London. His research interests include performance modelling, simulation, IoT, and edge analytics. Contact him at ranaof@cardiff.ac.uk.

Surya Nepal is a principal research scientist at CSIRO Data 61, Australia. His research interests include cloud computing, Big Data, and cybersecurity. Nepal has a PhD in computer science from Royal Melbourne Institute of Technology, Australia. Contact him at surya.nepal@csiro.au.

Mazin Yousif is the editor in chief of *IEEE Cloud Computing*. He's the chief technology officer and vice president of architecture for the Royal Dutch Shell Global account at T-Systems International. He has a PhD in computer engineering from Pennsylvania State University. Contact him at mazin@computer.org.

Philip James is a senior lecturer in the School of Civil Engineering and Geosciences at Newcastle University, UK. His research interests include the Internet of Things, next-generation analytics, and spatial data management. Contact him at philip.james@ncl.ac.uk.

Zhenyu Wen is research fellow at Newcastle University, UK. He received a PhD (2016) in Cloud Computing from Newcastle University. His research interests includes IoT, Distributed systems, Big Data Analytics and Computer network. Contact him at z.wen@ncl.ac.uk.

Stuart Barr is a professor in the School of Civil Engineering and Geosciences at Newcastle University, UK. He works on a range of cross-disciplinary problems in the field of Earth systems engineering. Contact him at stuart.barr@ncl.ac.uk.

Paul Watson is a professor of computer science at Newcastle University. His research interests include scalable computing systems, data-intensive problems, and cloud computing. Watson has a PhD in computer science from Manchester University. Contact him at paul.watson@newcastle.ac.uk.

Prem Prakash Jayaraman is a research fellow at the Swinburne University of Technology. His research interests include Internet of Things, cloud computing, big data analytics. Jayaraman has a PhD in computer science from Monash University. Contact him at prem.jayaraman@gmail.com.

Dimitrios Georgakopoulos is a professor at Swinburne University of Technology. His research interests include the Internet of Things, data management. Georgakopoulos has a PhD in computer science from the University of Houston, Texas. Contact him at dgeorgakopoulos@swin.edu.au.

Massimo Villari an associate professor of computer science at the University of Messina. His research interests include cloud computing, Internet of Things, big data analytics, and security systems. Villari has a PhD in computer engineering from the University of Messina. Contact him at mvillari@unime.it.

Maria Fazio is an assistant researcher in computer science at the University of Messina (Italy). Her main research interests include distributed systems and wireless communications. She has a PhD in advanced technologies for information engineering from the University of Messina. Contact her at mfazio@unime.it.

Saurabh Garg is currently working as a lecturer in the Department of Computing and Information Systems at the University of Tasmania, Hobart, Tasmania. He received a Ph.D. from the University of Melbourne in 2010. Contact him at Saurabh.Garg@utas.edu.au.

Rajkumar Buyya is a professor of computer science and software engineering, at the University of Melbourne, Australia. His research interests include cloud, grid, distributed, and parallel computing. Buyya has a PhD in computer science from Monash University. Contact him at rbuyya@unimelb.edu.au.

Lizhe Wang is a professor at the Institute for Remote Sensing and Digital Earth, Chinese Academy of Sciences, and at the School of Computer at the China University of Geosciences. Wang has a PhD in computer science from Karlsruhe University. Contact him at lizhewang@icloud.com.

Albert Y. Zomaya is the Chair Professor of High Performance Computing & Networking at the University of Sydney. His research interests include parallel and distributed computing, and data intensive computing. Zomaya has a PhD from Sheffield University. Contact him at albert.zomaya @sydney.edu.au.

Schahram Dustdar is a full professor of computer science heading the Distributed Systems Group at TU Wien, Austria. His work focuses on Internet technologies. He's an IEEE Fellow and a member of the Academy European. Contact him at dustdar@dsg.tuwien.ac.at.

Digital Transformation Through SaaS Multiclouds

Brad Power
MAXOS

Editor:
Joe Weinman,
joeweinman@gmail.com

There are lots of big companies that would love to switch from their big legacy systems to avoid compromises in functionality, make them more agile, lower IT costs, and help them to become faster to market. This article describes how they can make the move.

Many executives are frustrated by their rigid IT systems (ERP suites and packaged software) which can force compromises on functionality, may make them slow to respond to market changes, and can be expensive to support and maintain. In the 1990s and early 2000s, the golden age of ERP, many companies followed the conventional wisdom that they needed a "single source of truth." They believed the only way to achieve that was to buy a big, comprehensive suite of software applications from one vendor, such as SAP, Oracle, JD Edwards, or PeopleSoft. Today, pioneering companies are choosing instead to free managers to choose multiple "best of breed" cloud applications to assemble a SaaS multicloud. These early adopters are typically more agile and have lower IT costs than their competitors.

Cloud software has been around for a long time—for example, Salesforce, a pioneer of cloud software, is 20 years old. However, it's only recently that leading companies are shifting from tactically acquiring one or two cloud services on a piecemeal basis to strategically exploiting a SaaS multicloud strategy: accessing most of their software from cloud providers, assembling them like Lego blocks, and swapping them in and out of their IT architecture as needed. Instead of having one monolithic database, when a manager needs information, the source for the relevant data is clear, and a layer of reporting software seamlessly integrates heterogeneous data from the different sources.

GETTING SOFTWARE FROM MULTIPLE CLOUD PROVIDERS CAN ENABLE DRAMATIC COST REDUCTION

Allan Farrell, CIO at FCR Media Belgium, was being challenged by his new boss: Reduce annual IT spending from €14M to €5M. In 2016, FCR had acquired Farrell's company, Truvo, which had been based on the printed Yellow Pages directories that advertised small and medium businesses, and everyone could see that was a dying business model. Headcount in Belgium was slated to be reduced from 600 to 300. So, Farrell devised a strategy to achieve his aggressive IT

Copublished by the IEEE CS and IEEE ComSoc
2325-6095/18/$33.00 USD

budget reduction target without degrading the company's capabilities by relying on three principles: Cloud first; buy, don't build; and configure, don't customize. Software services were switched from a standard industry software package for billing, operations, and customer service, and another package for accounting and finance to 10 cloud applications acquired on the Salesforce AppExchange. In less than five months, he reduced IT costs from €14M to €6M and IT headcount from 75 to 20. Customer satisfaction improved by 50 percent. And in the latest list of the top digital marketing agencies in Belgium, FCR was ranked number two, even though it only launched in 2016.

GETTING SOFTWARE FROM MULTIPLE CLOUD PROVIDERS CAN ENABLE GROWTH

More than 30,000 small- and medium-sized businesses trust Paycor to deliver their payroll and human resources services. For its first 20 years, Paycor grew in the single digits every year, but about five years ago the company decided to accelerate its annual growth to 20-25 percent. To grow that fast, it needed to upgrade its systems for marketing, sales, and customer service. Brian Vass, vice president of sales and marketing technology, told me that its IT organization is good at building payroll and human resources systems, but "it isn't good at these other systems, and even if they wanted to develop them, progress would move at a snail's pace." Paycor decided to make a fundamental shift from inhouse to SaaS for its new software. It selected two major cloud products—Salesforce and Marketo—and added 40 compatible apps from the app store of its major provider—Salesforce AppExchange.

Vass said, "We wouldn't be able to move this fast without a SaaS model. We can find a product that meets our needs, and with one click we're up and running in minutes."

And everything works together. Paycor is growing at a pace that would have been impossible without leveraging multiple cloud services.

THE NEW MULTICLOUD ARCHITECTURE

In my interviews with 10 companies that have chosen a "plug and play" applications architecture utilizing multiple cloud software providers, a few key design principles emerged:

Process First

Switch from being forced to follow the practices embedded in one big software package to shopping for smaller software modules that fit your ideal process, and if none are available due to unique process requirements, then build your own. For example, Joe Cardella, senior vice president of sales at waste removal company Cardella Waste Services, switched from a standard industry software package that didn't quite fit the company's needs to six cloud apps, including a real-time operations management system, to get software services that fit better with its unique business needs—tracking trucks and dumpsters.

Multiple Databases

Shift from a "single source of truth" in a monolithic database to multiple "master databases", e.g., one each for customers, human resources, finance, and operations, sourced from clearly designated applications. For example, Erik den Ouden, the director of global IT for JF Hillebrand, a logistics company, pointed out that the company uses one cloud application as the source for customer management, sales, and marketing data; another for finance; and a third for human resources. It also has a homegrown system for order management because no products meet the unique needs of its business. Each application is the source for data in that function ("system of record"). An integration layer facilitates customer and employee interactions. When a manager needs information, the source for the relevant data is clear, and a layer of reporting software brings the data together seamlessly from the different sources.

Experiment and Swap

Continuously run experiments to find better solutions. For example, Charlie Merrow, CEO of Merrow Manufacturing, a high-tech sewing machine and soft goods manufacturer, told me they selected a shipping application that was the best on the market, then they switched to a UPS product, then, when their original provider upgraded, returned to the original application. This was made possible by having small, independent applications that easily communicate with the other applications.

STRATEGIES FOR DIGITAL TRANSFORMATION TO SAAS MULTICLOUD

There are lots of companies that would love to transition from their legacy, monolithic ERP systems to a SaaS multicloud approach to avoid compromises in functionality, increase business agility, lower IT costs, and reduce time to market. But there's not much guidance on how to move from legacy suites to multiple cloud applications. However, key lessons have emerged to help manage such digital transformations:

Starve the Beast

Mike Gibbons, CIO of Aggregate Industries, a producer of asphalt, aggregates, and ready-mixed concrete, gradually replaced parts of its ERP solution with multiple cloud solutions. The ERP system was close to their needs with some customization, but as their business evolved and each set of new requirements emerged they first looked for a cloud solution that at least met their custom internal requirements. Over time, they introduced Anaplan for forecasting, ServiceNow for IT services, Salesforce for customer management, Workday for HR, Coupa for procurement, Qlik for business intelligence, and Google Enterprise for office productivity—which saved them thousands of dollars. While ERP implementations typically take years, these smaller cloud application implementations were much faster. For example, the Coupa replacement of all the procurement functionality took less than three months. They could do it that fast because they worked out the simplest process first, and it was in a bite-sized chunk. Gibbon said, "We got better technology and better results."

Spin It up Fast, Throw It Away

Merrow Manufacturing's technology strategy is to be iterative, be fast, and eliminate things that don't work. They leverage technology to grow and empower everyone to be more commercially successful. Sometimes they find a cloud application that meets their needs, but they also aren't afraid to develop and implement their own technology to do what they need. For example, their phone system had issues, so they developed one themselves. It's not just about cost savings: Merrow can now customize and tune it to meet their own requirements and align with their processes. Adjusting technology to fit their processes is something they've become good at. They minimize the number of times that data needs to be entered, or that someone needs to remember something that needs to be done. Their software is running at a low cost and with little time invested, so they can discard an application if it doesn't fit and develop their own using platform services. For example, they needed an application for managing sewing machine repairs. There was nothing like this on the Salesforce AppExchange. So, they documented their requirements, including the reports they wanted to run. Then they built the system on the Lightning Platform and were up and running in two days. It wasn't exactly right, but they used an agile approach and with business meetings every day providing feedback on reporting data and formats, they iteratively made it better. Then they moved to workloads. "What do we want to remind people about? How do we reduce the information flows to customers?" It fundamentally changed the way they delivered customer service around repairs. In very short order, this process went from taking too long to completely changing a part of their business that added net-new revenue. Gibbon said, "It didn't take a big investment in software. But it warranted enough investment to do it well."

Standardize Selectively

Christine Ashton, Global Chief Digital Officer at SAP in the Digital Office ERP Cloud, pointed out that the kind of agile development that Charlie Merrow advocates has its place, yet sometimes commoditization and standardization can be useful. For example, payments and compliance can be purchased from a provider of an industry standard, yielding services at competitive parity and making it easier for ecosystem partners to connect. For example, all the companies I interviewed had purchased software applications for finance and accounting, which represented up to 50 percent of their software. This selective use of commodity software requires a decomposition of work into a services and data architecture and a "heat map" of what needs to be commodity vs. distinctive.

THE ROLE OF SYSTEMS INTEGRATOR

Despite the benefits, a multicloud architecture means that a company must be its own systems integrator, rather than relying on the monolithic software package provider. As described by Erik den Ouden of JF Hillebrand, a company with a multicloud architecture needs to integrate data across applications, often using middleware or reporting software, instead of having most of the data in one database. Dependencies need to be more closely watched to ensure that compatibility and compliance are maintained. Rather than negotiating with just one or two cloud infrastructure, platform, and/or service providers, there now may be dozens. Different interfaces may create challenges for user training or usage.

Allan Farrell of FCR Media Belgium focuses his IT staff on architecture and vendor management, while relying on high quality partners for the rest of their IT services, such as IT operations out of India, a big systems integrator for Salesforce configuration, and another partner for media content. One of his challenges was finding the right kind of business architect to work with the business. He found an excellent one who did a great job of solving the pain points on the customer journey by mapping to capabilities. It took some time to shift the business managers from seeing everything as an IT problem to start with business capabilities and business processes. The business architect now reports to the CEO on strategy and innovation.

CONCLUSION

The multicloud software architectures of the companies profiled in this article using more, smaller applications have many advantages: lower cost, growth, and best-of-breed services. Yet these companies are a rare breed. Allan Farrell of FCR Media said that he hasn't come across anybody else who has done what they've done in the time they've done it, and Salesforce and Deloitte haven't either. Chris Meyerpeter, CIO at Ardent Mills, a $4 billion flour supplier, which has had a "Velcro" architecture of over 40 cloud applications for four years, said he hasn't seen anyone else doing this at their scale. It is clear, however, that as companies in all industries undergo digital transformation, a SaaS multicloud approach has numerous benefits.

ABOUT THE AUTHOR

Brad Power is a consultant who helps organizations that must make faster changes to their products, services, and systems to compete with start-ups and leading software companies. He is a principal at MAXOS, a partner at FCB Partners, a Questrom Digital Fellow at Boston University, an advisor to the Innovation Scout, and a frequent contributor to the Harvard Business Review. He received a BS in Mathematical Sciences from Stanford University and an MBA from UCLA. Contact him at bradfordpower@gmail.com.

Contact department editor Joe Weinman at joeweinman@gmail.com.

Cloud Reliability: Possible Sources of Security and Legal Issues?

Marcello Cinque
University of Napoli "Federico II"

Stefano Russo
University of Napoli "Federico II"

Christian Esposito
University of Napoli "Federico II"

Kim-Kwang Raymond Choo
University of Texas at San Antonio

Frederica Free-Nelson
Army Research Laboratory

Charles A. Kamhoua
Army Research Laboratory

Editor:
Kim-Kwang Raymond Choo
raymond.choo@fulbrightmail.org

Cloud computing is a key supporting technology driving the fourth industrial revolution, spanning from the Internet of Things (IoT) and Cloud for Telecoms (C4T) to Industry 4.0 to Smart Cities. This pervasiveness of cloud technology is due to its ability to easily share and obtain resources on a pay-per-use and elastic provisioning model. Several enterprises use the cloud as a cheap solution to achieve computational and storage capabilities they need without incurring in the costs associated with owning and maintaining data centers. This implies that cloud-based applications rely on the correctness of the services provided by the cloud platforms and that any possible outage could result in considerable loss of reputation and money to the application provider. This could also effectively bring actions against the cloud service provider for violations to the agreed Service Level Agreement (SLA).

It is crucial to investigate cloud resiliency and ensure that we have the means to provide a suitable level of resiliency to avoid potential Service Level Agreement (SLA) violations. In fact, cloud outages have an impact on enterprise workloads or popular consumer applications. The bigger the cloud service provider, the higher the economic consequences of the outage. Within the literature, there is no agreement on the definition of resiliency, and in most cases, it is assumed as synonymous for reliability or survivability. Resiliency can be defined as the ability of a

Copublished by the IEEE CS and IEEE ComSoc
2325-6095/18/$33.00 USD ©2018 IEEE

system to keep its application running by keeping the right level of quality, despite any unexpected variations in the working conditions, such as applied workload, experienced fault load, or successful attacks.[1] Under the umbrella of resiliency, we can find the set of secondary properties composed of fault-tolerance (i.e., keeping the normal conditions by tolerating the occurrence of any faults), survivability (i.e., preservation of essential assets for a system despite of large-scale disasters), security (i.e., the preservation of the correct behavior from possible threats and attacks to the cloud infrastructure), scalability (i.e., the ability of the system to perform well despite of excessive workloads and execution conditions). Despite the fact that all of these properties are evenly important, fault-tolerance is particularly crucial for cloud computing, due to its complexity and the ineffectiveness of traditional solutions. Additionally, depending on the resource-provisioning model, we can have different fault models and methods to cope with these faults.

Cloud computing has three different provisioning models, namely:

1. Infrastructure-as-a-Service (IaaS), which consists of making physical computing resources and virtual machines (VM) accessible through the Internet,
2. Platform-as-a-service (PaaS), which makes available to the users frameworks and environments to build and deploy applications, such as Web servers, databases and so on,
3. Software-as-a-Service (SaaS), which implies hosting application software within the cloud and make it available on a subscription basis.

Given the model being applied, we can schematically abstract the internals of a cloud platform as in Figure 1, where we have a set of physical computing resources interconnected among themselves by a Local Area Network (LAN) and hosted within multiple datacenters owned by a single cloud service provider. These datacenters can be geographically distributed and federated by means of a Wide Area Network (WAN); however, the user is not aware of such a distributed implementation of the platform thanks to a front-end. In other words, one or multiple machines forward the incoming resources to the right datacenter based on the relevant heuristics. Within each physical machine, we can find multiple VMs running and managed by a proper hypervisor and virtualization platform, where services and applications execute. Many kinds of faults can happen within a cloud infrastructure, and may cause failures compromising the services hosted within the cloud and perceived by the users.[2]

Without going into details for the specific fault types, we can briefly assume that the physical layer hidden by the IaaS model can be affected by faults at the hardware of the physical nodes composing the data center, and/or the networking devices composing the local network within the data center. These faults may manifest due to the unavailability of certain physical computing machines, or their massively degraded performance, while we can experience several networking misbehaviors (e.g., message losses, reordering, corruption or delayed delivery) whose occurrence probability increases more than usual. As a side effect of these failures, resources at higher levels can be compromised. For example, if a physical device fails, we can have a cascading effect on the hosted virtual resources, as well as on the services and applications contained in such virtual resources.

Programming mistakes, misconfigurations or even non-fatal hardware errors can trigger erroneous behavior of the software elements within the cloud. These problems can first involve the used virtualization means to realize the IaaS, such as the VM Monitor (e.g., the Xen hypervisor), and may cause failures, ranging from simply detectable crashes to unpredictable and erratic runtime behaviors. The effects of these failures are not limited only to IaaS, but can also propagate to the services built on top of it, such as PaaS and SaaS. Software faults can also occur at the higher abstraction levels, such as at the integration middleware and its programming interface for the VM management, the hosted Web server or databases, and the applications exposed as SaaS. The causes of these faults can be related to human errors, bugs, misconfigurations or even natural and human-induced disasters. The failures caused by these faults can be due to the provided services/applications being affected by disruptions, degraded quality-of-service (in terms of performance, or availability), and/or corrupted data and functionalities. Most of these failures can be perceived by the users as a violation of the contractually-agreed SLA and might trigger legal actions with consequent economic and reputation losses for the cloud service provider.

With organizations moving to the cloud, especially for military defense, it is imperative to minimize some of the disruption, low performance, and availability. Therefore, it is important to ensure that the cloud platforms have the means to tolerate such faults. The research community has investigated such means drawn from the dependability literature, and adapted to the peculiarities of cloud computing, so that current cloud service providers can protect their solutions from possible faults and the consequent failure effects.

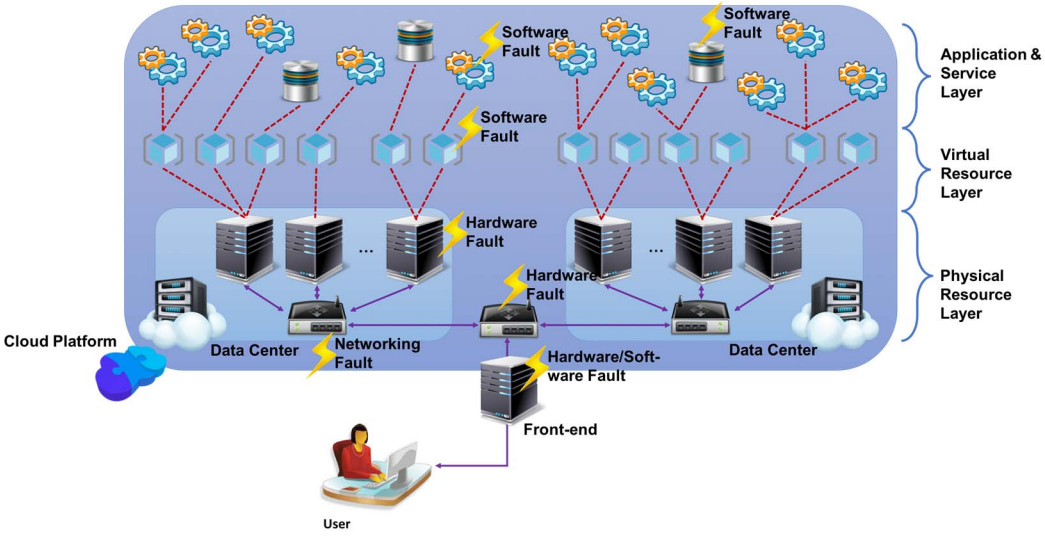


Figure 1. Constituents of a cloud platform and potential faults.

MEANS TO COPE WITH FAILURES IN CLOUD INFRASTRUCTURES

Several techniques can be integrated to provide a suitable level of fault tolerance at various levels of a cloud platform. Figure 2 provides a schematic taxonomy of fault-tolerance solutions.

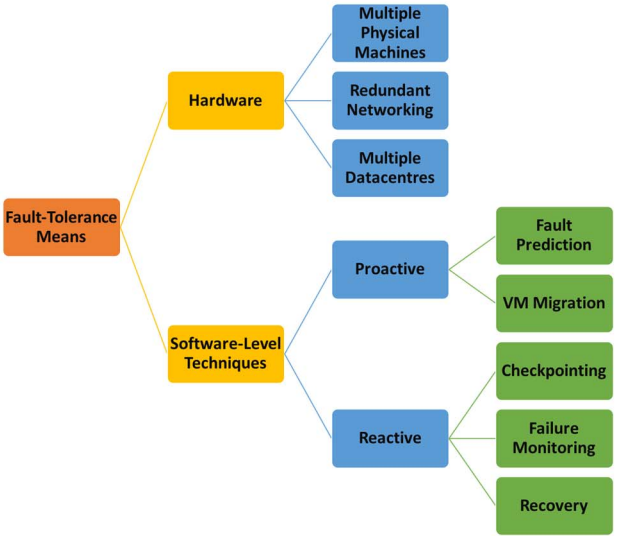


Figure 2. Taxonomy of the fault-tolerance solutions.

Architectural redundancy is the key approach to cope with hardware faults in a cloud data center.[2] For instance, having multiple physical nodes allows managing failover from a failed node to a standby node; using RAID for disk storage within increases data availability; having redundant network equipment and link supports reliable communications, where the failures of the network link or device do not affect correct communications.

Hardware redundancy alone is not sufficient, but it is necessary to couple it with proper software techniques such as machine learning algorithms built specifically to deal with failures. Such techniques can be grouped in proactive and reactive ones.[3] In both cases, the resources within the platforms need to be monitored, but with different intentions. In the first case, the objective is to predict a possible fault so as to trigger a way to prevent a failure within the platform. In the second case, the monitoring aims at detecting a possible failure so that recovery actions can be triggered. The proactive monitoring is typically coupled with live VM migration[7,13]: before the manifestation of a fault, a VM is moved towards a novel physical device within the same cluster or datacenter (encompassing multiple clusters). Such a migration is done without previously stopping the VM, but while it is servicing its scope, and may involve not only the VM, as well as applications and services. In particular, the VM state is saved and transferred over the network to a newly activated VM on another physical device. Several platforms do not recur to a single data center, but they have several data centers around the world for performance reasons, so that a request can be served by the geographically nearest datacenter and reduce the latency. For example, Amazon has reportedly fragmented the world into 16 regions where there is more than one datacenter (e.g., in Europe there are around 16 datacenters servicing the European customers and the ones in the Middle East) with at least 30 datacenters operated in total. When there are multiple datacenters, live VM migration can be undertaken among them. The migration of virtual resources and services among datacenters has proved to be a more powerful solution than within a same cluster or datacenter to achieve fault tolerance,[6] so as to remove the probability of common-mode failures and/or tolerating natural/man-made disasters. Many of the existing virtualization tools, such as Xen, support live migration. Generally speaking, this migration implies considerable costs, despite research efforts on reducing them. In addition, fault prediction is still an open challenge since the current forecasting solutions exhibit a low accuracy and precision.[8]

The reactive solutions are based on the combined use of three techniques[2,3]: checkpointing (i.e., the activity of periodically saving the state of a VM or even service/applications), failure monitoring (i.e., the continuous monitoring a device or resource by testing its correct behavior), and recovery (i.e., substituting a failed resource by a new one whose stated is copied from the saved one of the failed resource). Such a solution is not only limited to VM and can be automatically provided by the cloud platforms. This is due to their auto scaling mechanisms, and is typically leveraged to dynamically scale up services and applications in response to increasing/lowering demand. Such a feature can also be used to substitute failed resources. However, this method has a considerable failover, caused by the delay to detect a failure and recover the failed resource.

VMs, as well as services and applications, can be replicated in an active and passive way[9] (so as to be classified as proactive in the first case and reactive in the latter one). Such a replication can be done automatically and seamlessly by the hypervisor, offering a proper Replication Management, the cloud platforms due to its auto scaling mechanism, or may depend on an explicit deployment plan or certain design of the application and server. Specifically, it is possible to have multiple instances of a given type of VM. In the first case, all of them are actively serving the incoming requests, while in the second case only a subset is used and the rest of them takes the place of the active ones after their failure.

Lastly, virtualization tools are currently able to provide diagnostic data on their behavior and/or providing migration and backup capabilities. In addition, there is ongoing research in investigating the major faults occurring within such software and proposing new designs to remove single-point-of-failures, allowing high degrees of fault-tolerance.[10]

VULNERABILITY OF FAULT-TOLERANCE SOLUTIONS

A holistic approach that considers the joint provisioning of fault-tolerance, resiliency and security is conventionally not available, as each of these non-functional properties, or even their secondary properties, are investigated separately. However, it is not surprising that a solution designed to support one of these properties may cause violation of the conditions to guarantee one or the remaining properties. In particular, it is interesting to note that the adoption of one of the aforementioned fault-tolerance means may cause issues with respect to the security and privacy of the cloud platform. This is because these solutions have been designed in isolation.

We will now discuss the privacy and security issues relating to fault tolerance in cloud computing (see also Figure 3), and the potential research opportunities.

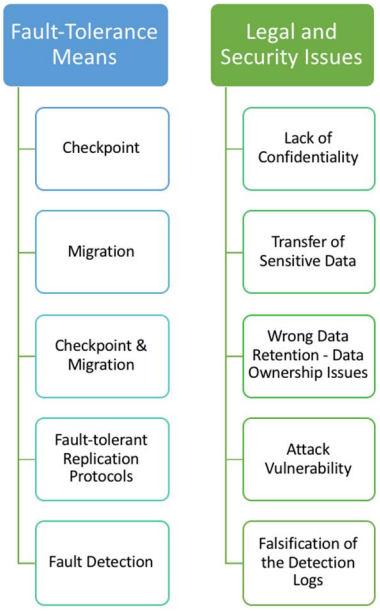


Figure 3. Relative issues of fault-tolerance means within cloud platforms.

A number of legal risks exist relating to cloud computing, apart from possible SLA violations caused by untolerated faults. Examples include issues related to data management.[11] Specifically, the cloud may host sensitive and personal data related to its users and customers of the hosted services and applications, whose privacy must be protected as mandated by various legal frameworks. One important requirement is for data to be protected against unauthorized disclosure, and cloud platforms are equipped with the means to avoid data leakages. However, state recording conducted by checkpointing may expose sensitive data or make checkpointed data vulnerable to possible disclosures and unauthorized access, including by malicious insiders. Migration toward alternative datacenters may imply the transfer of data to geographical areas subject to opposing legal guarantees from those where the data previously resided. Such transfers occur seamlessly and the cloud user may not be aware of them. The possibility of having data stored on a backup server, a checkpointing service or migrated to an alternative datacenter may compromise the correct data retention post termination of the SLA. When a relationship between a customer with a cloud service provider terminates, all of his/her data must be erased from the cloud platform. However, fault-tolerant mechanisms may exacerbate this operation since it is complex to determine where such data reside. Further, the hypervisor does not provide complete isolation among cloud VMs, which opens the possibility of side-channel attacks.

The risk is compounded if data are stored in multiple servers. The various legal frameworks have several principles controlling the ownership of data. The European directive on data privacy, for

example, has a clear definition of data ownership and spells out the obligations of data storage services. However, the fault-tolerance being deployed may violate such legal obligations, since, for example, the cloud service provider may not allow the user to gain access to back-end data represented by the back-up or checkpoint details for VMs migration or recovery.

A *possible research challenge* to be considered is how to support the right level of data retention, confidentiality and ownership when using checkpointing and migration. One could also consider using data encryption to resolve the issue, since without having access to the decryption key, another entity will not be able to decrypt the encrypted data even if the encrypted data is not under the control of the data owner. However, there are performance considerations, since the checkpointed data must be used to restore a failed VM. To address these security requirements, most government agencies rely on IT security standards and certification to assess cloud assurance.[14] However, none of the major standards fully address all potential vulnerabilities in the cloud.

To enhance the security of the cloud platform, one could embed the replication within the cloud platform in order to cope with possible vulnerability and attacks. In other words, it will be more difficult to compromise the availability and influence the (correct) behavior of the hosted resources[12]. Moreover, fault isolation has the side-effect of avoiding the propagation of the negative effect of an attack that have successfully compromised some resources within the cloud platform. However, it is important to consider that fault-tolerant replication protocols assume non-malicious faults. Hence, if the unavailability of certain resource is due to a successful attack, even if they have been replicated, there is no guarantee of having a high level of availability. Specifically, if an attack is successful on one resource, then it can be successfully conducted on the other duplicated or related resources, resulting in all these resources being unavailable.

A *possible research direction* is to investigate the use of replication diversity[15], typically adopted in safety critical application domains, also in the case of VM replication in terms of adopted infrastructures, including the hypervisor, the operating system, and the deployed security countermeasures (e.g., firewall, intrusion detection systems, etc.). This would help to minimize the probability of success of the same attack on diverse replicas.

Fault detection is crucial to trigger the relative recovery; however, there is a security implication. When there is an abnormal behavior of a service/application, with a consequent SLA violation, the culprit can be the cloud or the network between the cloud and the user. It is important to detect where the fault occurs so that the culprit can be identified and the appropriate action be undertaken (e.g. prosecution). A cloud service provider can find it beneficial to be able to game the measurements to their advantage so as not to take the responsibility of the occurred faults and consequent SLA violations. It is important to achieve secure fault detection, without the possibility for the cloud service provider, or its personnel, to falsify the logging audits and hide SLA violations. Similarly, an attacker can also exploit this vulnerability to eliminate their traces from the system. Thus, we need to design novel solutions to securely store logging data and to avoid their modification after some failure events or security attacks.[16]

REFERENCES

1. F. Longo et al., “An Approach for Resiliency Quantification of Large Scale Systems,” *SIGMETRICS Performance Evaluation Review*, vol. 44, no. 4, 2017, pp. 37–48.
2. C. Colman-Meixner et al., “A Survey on Resiliency Techniques in Cloud Computing Infrastructures and Applications,” *IEEE Communications Surveys & Tutorials*, vol. 18, no. 3, 2016, pp. 2244–2281.
3. M.A. Mukwevho and T. Celik, “Toward a Smart Cloud: A Review of Fault-tolerance Methods in Cloud Systems,” *IEEE Transactions on Services Computing*, 2018; doi.org/10.1109/TSC.2018.2816644.
4. M.N. Cheraghlou, A. Khadem-Zadeh, and M. Haghparast, “A Survey of Fault Tolerance Architecture in Cloud Computing,” *Journal of Network and Computer Applications*, vol. 81, no. 92, 2016.

5. A.M. Sampaio and J.G. Barbosa, "A Comparative Cost Analysis of Fault-Tolerance Mechanisms for Availability on the Cloud," *Sustainable Computing: Informatics and Systems*, 2017; doi.org/10.1016/j.suscom.2017.11.006.
6. R. Jhawar and V. Piuri, "Chapter 9 - Fault Tolerance and Resilience in Cloud Computing Environments," *Computer and Information Security Handbook*, John R. Vacca, Morgan Kaufmann, 2017.
7. R.W. Ahmad et al., "A survey on virtual machine migration and server consolidation frameworks for cloud data centers," *Journal of Network and Computer Applications*, vol. 52, no. C, 2015, pp. 11–25.
8. T. Hall et al., "A systematic literature review on fault prediction performance in software engineering," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, 2012, pp. 1276–1304.
9. M.A. Khoshkholghi et al., "Disaster recovery in cloud computing: A survey," *Computer and Information Science*, vol. 7, no. 4, 2014, p. 39.
10. J. Groß, "A Fault Tolerant Virtualization Server Based on Xen,"; https://events.static.linuxfound.org/sites/events/files/slides/Xen-Host.pdf.
11. K. Djemame et al., "Legal issues in clouds: towards a risk inventory," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 371, no. 1983, 2013.
12. A. Mazhar et al., "DROPS: Division and Replication of Data in the Cloud for Optimal Performance and Security," *IEEE Transactions on Cloud Computing*, vol. 6, no. 2, 2018, pp. 303–315.
13. M. Ficco et al., "Live Migration in Emerging Cloud Paradigms," *IEEE Cloud Computing*, vol. 3, no. 2, 2016, pp. 12–19.
14. C.D. Giulio et al., "IT Security and Privacy Standards in Comparison: Improving FedRAMP Authorization for Cloud Service Providers," *17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (CCGRID 17), 2017, pp. 1090–1099.
15. C.A. Kamhoua et al., "Survivability in Cyberspace using diverse replicas: A Game Theoretic Approach," *Journal of Information Warfare*, vol. 12, no. 2, 2013, p. 27.
16. M.A. Manazir et al., "CLASS: Cloud Log Assuring Soundness and Secrecy Scheme for Cloud Forensics," *IEEE Transactions on Sustainable Computing*, 2018; doi.org/10.1109/TSUSC.2018.2833502.

About the Authors

Marcello Cinque graduated with honors from the University of Naples with a PhD in computer engineering. He is an associate professor at the Department of Electrical Engineering and Information Technology of the University of Naples Federico II. Cinque is chair and/or TPC member of several technical conferences and workshops on dependable systems, including IEEE DSN, EDCC, and DEPEND. His interests include dependability assessment of critical systems and log-based failure analysis. Contact him at macinque@unina.it.

Stefano Russo is a professor of Computer Engineering at the University of Naples "Federico II" and leads the Dependable Systems and Software Engineering Research Team (DESSERT). He coauthored over 160 papers on software engineering, software dependability, middleware technologies, and mobile computing. He is an associate editor of the *IEEE Transactions on Services Computing* and senior member of IEEE. Contact him at stefano.russo@unina.it.

Christian Esposito is an assistant professor at the University of Naples "Federico II", where he received his PhD in computer engineering and automation. His research interests include reliable and secure communications, middleware, distributed systems, positioning systems, multi-objective optimization, and game theory. Esposito has served as a reviewer or guest editor for several international journals and conferences, and has been a PC member or organizer of about 40 international conferences/workshops. He has also served as guest editor for several journals, and is a member of three journal editorial boards. Contact him at christian.esposito@unina.it.

Kim-Kwang Raymond Choo holds the Cloud Technology Endowed Professorship in the Department of Information Systems and Cyber Security at the University of Texas at San

Antonio. His research interests include cyber and information security and digital forensics. He is a senior member of IEEE, a Fellow of the Australian Computer Society, and has a PhD in information security from Queensland University of Technology, Australia. Contact him at raymond.choo@fulbrightmail.org.

Frederica Free-Nelson is a researcher and program manager in Army Research Laboratory's Network Security Branch, where she leads research on machine learning and intrusion detection methods and techniques to promote cyber resilience and foster research on autonomous active cyber defense. Nelson currently serves as chairperson to the International Science Technology (IST-163) Panel –NATO Science & Technology Organization (STO) on the topic of Deep Machine Learning for military cyber defense. She is a participant in the Army Education Outreach Program as an ambassador and a virtual judge for the eCybermission program. Contact her at frederica.f.nelson.civ@mail.mil.

Charles A. Kamhoua is a researcher at the Network Security Branch (NSB) of the U.S. Army Research Laboratory (ARL) in Adelphi, MD, where he is responsible for conducting and directing basic research in the area of game theory applied to cyber security, cyber deception, and cyber physical system. He received a Ph.D. in Electrical Engineering from Florida International University. He is a senior member of IEEE. Contact him at charles.a.kamhoua.civ@mail.mil.

Cloud Reliability

Yury Izrailevsky
Consultant

Charlie Bell
Amazon

Cloud computing allows engineers to rapidly develop complex systems and deploy them continuously, at global scale. This can create unique reliability risks. Cloud infrastructure providers are constantly developing new ideas and incorporating them into their products and services in order to increase the reliability of their platforms. Cloud application developers can also employ a number of architectural techniques and operational best practices to further boost the availability of their systems.

Rapid adoption of cloud computing over the past decade left virtually no industry untouched: from video streaming (Netflix, Amazon Video) and gaming (Riot Games) to banking (Capital One), healthcare (Cerner) and services delivered by various government agencies (NASA, CIA, and the CDC). With more mission critical services running in the cloud, the expectations for cloud reliability are higher than ever. In this introduction, we will identify some of the common design principles and operational best practices that are being successfully employed to increase cloud service reliability, and illustrate them with examples.

RELIABILITY VS. COMPLEXITY AND THE RATE OF CHANGE

Cloud computing offers significant benefits in terms of scalability and performance. Elasticity of the cloud gives engineers the ability to rapidly deploy vast amounts of compute and storage resources. A global network of cloud regions enables low latency and interactive access to cloud-based services for customers around the world. At the same time, the vast array of edge locations brings content even closer through its delivery networks. Developer productivity is also significantly accelerated in the cloud. API-driven infrastructure and platform services, from core networking and storage components to advanced AI capabilities such as automatic speech-to-text (Amazon Transcribe) and image recognition (Google Cloud Vision API), enable engineers to integrate complex functionality into their applications with minimal development effort. Hosted continuous integration/continuous deployment (CI/CD) services and tools, such as Jenkins and Spinnaker, drastically reduce the amount of time it takes to push changes into production.

Engineers can use all of these components and services, cloud "building blocks," to develop complex systems and deploy them globally, faster than before. An online service may consist of hundreds of microservices, each running in its own distributed cluster and containing multiple dependencies of its own, increasing the number of individual components that can fail. Rapid

Copublished by the IEEE CS and IEEE ComSoc
2325-6095/18/$33.00 USD ©2018 IEEE

innovation is enabled by frequent changes in the code, configuration, and data. Yet each change also carries the risk of triggering a failure, potentially resulting in a customer facing outage.

Fault tolerant design principles and operational best practices can mitigate these reliability risks. Used properly, they can enable engineers to build "self-healing" systems that can mitigate failures without requiring human intervention and no-to-minimal impact on the users of these systems.

ARCHITECTING FOR FAILURE

On December 24, 2012, Netflix experienced an 18-hour outage affecting the majority of its customers.[1] It resulted from a regional failure of the AWS Elastic Load Balancer (ELB) service in the US-East-1 cloud region, where Netflix had all of its cloud operations at that time. Subsequently, AWS made significant improvements to its networking control plane, ELB service architecture and configuration management process.[2] Also learning from this outage, Netflix moved to an Active-Active cross regional architecture, where customer traffic is load balanced across three cloud regions, and data is kept in an eventually consistent sync using Cassandra (a distributed, cross-regional NoSQL database) replication.[3] In case of an outage, traffic from the affected region can be redirected to the remaining healthy regions, as Figure 1 shows.

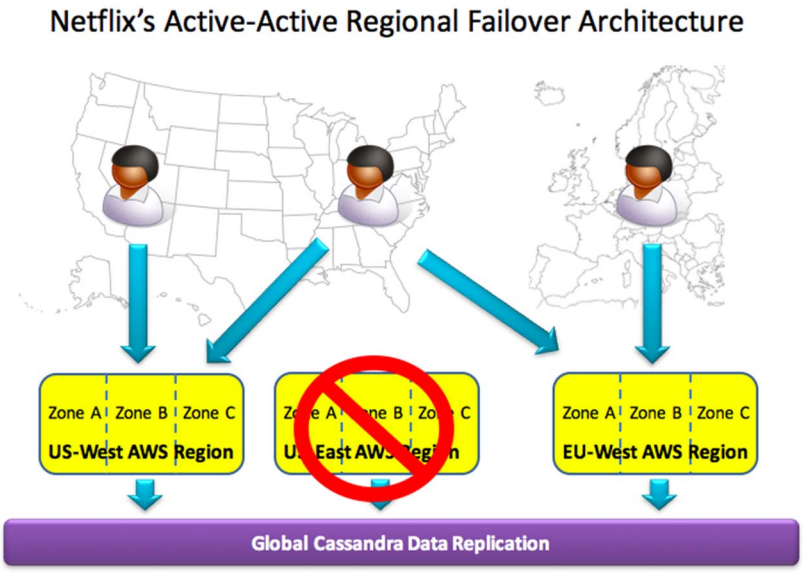


Figure 1. In the Netflix active-active cross regional architecture, customer traffic is load balanced across three cloud regions, and data is kept in an eventually consistent sync using Cassandra (a distributed, cross-regional NoSQL database) replication. In case of an outage, traffic from the affected region can be redirected to the remaining healthy regions.

Cloud infrastructure providers continue to invest significant resources toward increasing the resiliency of their services, both at the hardware and software levels. AWS added DynamoDB global tables and multi-master Aurora SQL engines, Azure is previewing multi-master support in Cosmos, and Google developed Spanner, a global quasi-relational database service that uses atomic clocks to guarantee strong consistency.[4] A paper featured in this special edition, "Virtual Machine Resiliency Management System for the Cloud," describes two complementary mechanism boosting virtual machine resiliency in IBM enterprise cloud managed service.

It is the goal of every cloud infrastructure platform to provide highly reliable components and services, but occasional failures are inevitable in any complex distributed system. While it has been demonstrated that the overall availability of a system is determined by the sum of its critical components,[5] it is possible to reduce the criticality of infrastructure dependencies by embracing

fault tolerant architectural principles, such as *redundancy* and *isolation.* Cloud application developers can employ well established design patterns, such as circuit breaker[6] and bulkhead,[7] to build systems that are resilient to failures of one or several underlying components. In essence, cloud provides infrastructure building blocks, but it is the responsibility of the application developers to stack them up and operate them in a way that maximizes the overall system availability.

Consider Netflix's current cross-regional architecture. Three cloud regions provide redundant capacity: if one of them fails, the other two can continue serving customer traffic from the affected region. In order to avoid global outages, changes must be isolated and deployed to one region at a time. Using libraries such as Hystrix,[8] which implements circuit breaker and bulkhead design patterns, allows to isolate and contain the "blast radius" of the failures of sub-components (such as individual microservices). From the cloud-provider side, AWS cloud regions utilize separate, independent installations of services, such as ELB, S3 (object storage), SQS (queueing), Auto-Scaling, and EBS (block storage). These regions also obey an independent deployment rigor, including immediate escalation to top leadership if an engineer creates a deployment pipeline with the capability of making simultaneous changes in multiple regions. When an outage hit S3 on February 28th, 2017, the issue was confined to a single region, allowing Netflix to continue operations with virtually no impact on its customers.

OPERATIONAL BEST PRACTICES AND CHAOS ENGINEERING

Any experienced DevOps engineer will tell you that there is no such a thing as a "safe" code deployment or configuration update: any change you make in production carries the risk of an outage. To minimize such risks, two techniques have been found to be particularly effective: blue-green deployments and canary releases.

With blue-green deployments, you set up two parallel production environments: one running the old version of the code (blue) and one running the new release (green), each capable of handling the full production workload. As traffic is routed from blue to green, the blue cluster remains fully operational, and if any problems are encountered with the new release, traffic is immediately rerouted back to the blue cluster. Once there is enough confidence in the new code, the blue cluster is spun down. This is a very powerful technique, which is enabled by the elasticity the cloud, where for a short period of time (e.g., a few hours) you effectively double your production footprint, and release the extra capacity when it is no longer needed.

Similarly, with canary releases, you set up a parallel "canary" environment running a new version of the code, but you only route a small subset of customers to it (1% is typical) while collecting performance and operational analytics, such as CPU/memory usage and error rates. This enables engineers to detect operational problems with the new release without exposing it to the full set of customers, and generally requires fewer additional cloud resources. Kayenta is an automated canary analysis tool integrated within the Spinnaker continuous delivery platform, which pulls user-defined metrics, runs statistical tests and generates an aggregate score for each canary release.[9] Typically, one runs canary analysis first, and if successful, proceeds to the full deployment using the blue-green mechanism.

To ensure the overall resiliency of the system, it is critical to have good visibility into the state of the system at all times in order to detect a failure quickly, and ways to mitigate the failure once it occurs. Tools such as CloudWatch[10] and Stackdriver[11] provided by cloud infrastructure vendors, and open source solutions such as Atlas,[12] enable engineers to instrument operational metrics in a scalable way, and trigger alerts based on any number of possible failure scenarios.

Incorporating fault-tolerant architectural principles and following deployment and monitoring best practices can significantly reduce the likelihood of production failures, but will not eliminate them completely. A live production environment, containing complex infrastructure and application dependencies and constantly stressed by ever-changing usage patterns (such as spikes in traffic or faulty network conditions), will expose a number of bottlenecks and failure scenarios over time that were never anticipated in the design, development, testing, and deployment phases. To identify and address such "ticking time bombs" proactively, a new discipline called Chaos

Engineering has emerged.[13] Chaos Engineering is focused on experimenting on a distributed system in order to build confidence in the system's capability to withstand turbulent conditions in production.

In 2011, Netflix announced its Simian Army,[14] which is a set of operational "Chaos" tools that trigger controlled failures in a live production environment, in order to ensure that the system can detect and automatically recover from such a failure, and if necessary, the engineering team is able to provide a timely and effective response. For example, Chaos Monkey randomly terminates live production instances, testing hardware redundancy. Latency Monkey injects additional latency into the client-server communication layer, testing the system's isolation and graceful degradation capabilities. Chaos Kong evacuates an entire cloud region, testing cross-regional failover capabilities in case of a catastrophic regional outage. These and many other types of tools have been developed over the years, and there is an active open source community developing and sharing Chaos tools.

The Chaos Engineering methodology, originally popularized by Netflix, has been embraced by a number of enterprises. However, many business owners have been reluctant to follow the approach, concerned about the associated costs, such as the impact Chaos production experiments might have on customer experience. In a paper included in this special edition, "The Business Case for Chaos Engineering," a group of Netflix engineers provide quantitative and qualitative reasoning for the necessity and cost-efficiency of the Chaos Engineering approach.

RECENT TRENDS ENHANCING CLOUD RELIABILITY

AI has been driving recent innovations across many different technology areas, and cloud reliability is no exception. Production environment of a modern enterprise, such as that of Amazon or Netflix, can generate over a billion operational metrics every minute;[12] runtime logs, aggregated over time, can scale into petabytes. It is simply impossible for humans to manually process such enormous volumes of data in order to identify small relevant bits, such as unique, non-obvious combinations of metrics that may lead to customer facing outages over time. Instead, supervised and unsupervised deep learning techniques, in combination with statistical methods, can be used to identify and help mitigate failure patterns.

Similar ML-based anomaly detection approaches can be applied to enterprise security (malicious insider behavior or intrusion detection), resource management (predictive auto-scaling of clusters in a cloud environment) and other problems. ML engines such as TensorFlow and MxNet, coupled with powerful GPU-powered hardware, such as NVIDIA Volta architecture and the P instance family from AWS, make deep learning capabilities readily available to a wide variety of cloud customers.

Rapid rise in the level of compute abstraction has been another significant development in the last couple of decades that has revolutionized the way companies manage their infrastructure. At the turn of the century, most production environments were deployed directly onto physical hardware instances, or "bare metal." Adoption of virtualization, moving to the cloud and architecting around container-based frameworks have successively allowed for a much more efficient use of hardware and supported dynamic workloads that can adapt to customer demand, survive hardware failures and reduce vertical scalability constraints of a physical instance. All of these capabilities have brought significant benefits in terms of reliability and operational efficiency, but they still require engineers to provision the underlying compute capacity, whether physical hardware on-premise or virtual instances in the cloud.

The emergence of serverless computing in recent years have propelled architectural and operational capabilities to a completely new level. The term "serverless" refers to a broad range of cloud capabilities and services, enabling developers to build complex systems by accessing prepacked applications and components, and allowing them to deploy custom application logic, while outsourcing most operational concerns such as "resource provisioning, monitoring, maintenance, scalability, and fault-tolerance to the cloud provider."[15] A powerful serverless paradigm called Function-as-a-Service (FaaS) in particular has been gaining rapid momentum. All major cloud vendors feature a FaaS offering, including AWS Lambda, Google Cloud Functions,

Microsoft Azure Functions and IBM OpenWhisk. FaaS enables developers to encapsulate custom logic within stateless ephemeral code blocks, or functions, which can be invoked both synchronously and asynchronously, triggered by a wide range of conditions and events, and whose resource provisioning and execution environment are managed entirely by the cloud provider.

Serverless computing is still in its early days, but cloud vendors are rapidly integrating serverless functionality into their platforms, making it a central part of their offering, and supporting an ever-increasing number of applications' use cases. Cloud customers, free from the constraints of resource provisioning and management, can focus on their applications' business logic, while their systems can operate more efficiently and reliably in a centrally managed, tightly monitored elastic pool of serverless capacity.

CONCLUSION

In this short introduction, we have summarized some of the architectural techniques and operational best practices that can be employed to increase cloud service reliability. Two selected articles show how reliability can be improved at the infrastructure level by boosting VM resiliency to failures, and at the application level by embracing the principles of Chaos Engineering. AI capabilities offer more promise in increasing cloud reliability in the future.

ACKNOWLEDGEMENTS

We would like to acknowledge the support from Dr. Mazin Yousif, the Editor-in-Chief of *IEEE Cloud Computing* magazine, and Brian Brannon, editorial content manager.

REFERENCES

1. A. Cockcroft, "A Closer Look at the Christmas Eve Outage," *Netflix Technology Blog*, blog, Netflix, 30 December 2012; https://medium.com/netflix-techblog/a-closer-look-at-the-christmas-eve-outage-d7b409a529ee.
2. "Summary of the December 24, 2012 Amazon ELB Service Event in the US-East Region," *AWS Service Message*; https://aws.amazon.com/message/680587/.
3. R. Meshenberg, N. Gopalani, and L. Kosewski, "Active-Active for Multi-Regional Resiliency," *Netflix Technology Blog*, blog, Netflix, 12 February 2013; https://medium.com/netflix-techblog/active-active-for-multi-regional-resiliency-c47719f6685b.
4. E. Brewer, "Spanner, TrueTime and the CAP Theorem," Google Research, 2017; https://storage.googleapis.com/pub-tools-public-publication-data/pdf/45855.pdf.
5. B. Treynor et al., "The Calculus of Service Availability," *Communications of the ACM*, vol. 60, no. 9, 2017, pp. 42–47.
6. M. Fowler, "CircuitBreaker," 6 March 2014; https://martinfowler.com/bliki/CircuitBreaker.html.
7. C. Bennage and M. Wasson, "Cloud Design Patterns," Microsoft Azure, 28 November 2017.
8. M. Jacobs, "How it Works," GitHub, 3 July 2017; https://github.com/Netflix/Hystrix/wiki/How-it-Works.
9. N. Kaul, "Introducing Kayenta: An open automated canary analysis tool from Google and Netflix," *Google Cloud Platform Blog*, blog, Google, 10 April 2018; https://cloudplatform.googleblog.com/2018/04/introducing-Kayenta-an-open-automated-canary-analysis-tool-from-Google-and-Netflix.html.
10. "Amazon CloudWatch," Amazon; https://aws.amazon.com/cloudwatch/.
11. "Google Stackdriver," Google Cloud; https://cloud.google.com/stackdriver/.
12. "Introducing Atlas: Netflix's Primary Telemetry Platform," *Netflix Technology Blog*, blog, Netflix, 12 December 2014; https://medium.com/netflix-techblog/introducing-atlas-netflixs-primary-telemetry-platform-bd31f4d8ed9a.
13. C. Rosenthal et al., *Chaos Engineering*, O'Reiley Media, 2017.

14. "The Netflix Simian Army," *Netflix Technology Blog*, blog, 11 July 2011; http://techblog.netflix.com/2011/07/netflix-simian-army.html.
15. I. Baldini et al., "Serverless Computing: Current Trends and Open Problems," *Research Advances in Cloud Computing*, Chaudhary S., Somani G., Buyya R., Springer, 2017.

ABOUT THE AUTHORS

Yury Izrailevsky advises venture capital firms and technology companies to help them optimize their cloud strategy and scale their engineering organizations. He holds BS and MS degrees in Computer Science from the University of Utah. Izrailevsky's research interests include distributed systems, cloud architecture and serverless computing. Contact him at izrailev@gmail.com.

Charlie Bell is Senior Vice President for AWS Products and Engineering. He has held this position since 2006 and served in various other technical leadership roles at Amazon since joining the company in 1998. He holds a BA in Business/MIS from California State University Fullerton. Bell's research interests range from specialized database systems to large-scale distributed systems reliability. Contact him at cbell@amazon.com.

The Business Case for Chaos Engineering

Haley Tucker
Netflix

Lorin Hochstein
Netflix

Nora Jones
Netflix

Ali Basiri
Netflix

Casey Rosenthal
Netflix

While Chaos Engineering has gained currency in the Site Reliability Engineering community, service and business owners are often nervous about experimenting in production. Proving the benefits of Chaos Engineering to these stakeholders before implementing a program can be challenging. We present the business case for Chaos Engineering, through both qualitative and quantitative tactics, as well as the benefits and tools to convince stakeholders this is necessary and cost-efficient.

THE DISCIPLINE OF CHAOS ENGINEERING

Chaos Engineering is the discipline of experimenting on a distributed system in order to build confidence in the system's capability to withstand turbulent conditions in production.[1] By running chaos experiments directly on a production system in a controlled manner, you can improve your system's availability by identifying and eliminating problems before they manifest as outages.

Consider a chaos experiment on a microservice architecture that injects latency between calls from service A to service B. This experiment might reveal that a spike in the response time of service B triggers a retry storm that overloads essential service C, a weakness that would not be detected by unit or integration testing (see sidebar: how chaos relates to traditional testing).

As an example, at Netflix there is an internal microservice that acts as a generic key-value store, and is used by other microservices. This store is not considered a critical service. We ran a chaos experiment where we intentionally injected latency for a small fraction of traffic so that calls to this service would exceed their timeouts and trigger fallback behavior. This experiment led to an unexpected increase in errors in a service that determines which content delivery network URLs to return to client devices, which is critical to video streaming. The exposed vulnerability was eliminated by adjusting timeout thresholds.

Copublished by the IEEE CS and IEEE ComSoc
2325-6095/18/$33.00 USD

This type of experimentation requires a technical shift within the organization. If applications were not originally designed to support chaos experiments, engineers must incorporate new tooling such as fault injection and guard rails to minimize blast radius[1]. More importantly, Chaos Engineering may require a cultural shift. In doing so, a successful program can change the way software engineers build systems by creating incentives for resilient design.

Chaos Engineering sounds dangerous ("you want to *intentionally* cause problems in the production system?"), so you'll need to make the case to your organization that a chaos program is worthwhile.

GETTING BUY-IN

The "It Doesn't Apply to Me" Fallacy

In some domains, such as finance, system failures can be extremely costly. The Knight Capital Group, a U.S. trading firm, famously lost over $400 million because of a software configuration issue.[2] In other domains, such as transportation, a system failure can lead to injury or loss of life. It is not surprising that stakeholders in these domains would be reluctant to run experiments on production systems.

Yet, it is because of the increased cost of impact that we believe Chaos Engineering should have a higher focus in these types of organizations. Regardless of the level of impact, it's important to have safety features with chaos experiments that minimize blast radius. Experimenting in production has the potential to cause unnecessary customer pain. While there must be an allowance for some short-term negative impact, it is the responsibility and obligation of the Chaos Engineering team to ensure the fallout from experiments is minimized and contained.[1] If Chaos Engineering is well-scoped, the return on investment (ROI) of the practice outweighs the short-term pain it can cause along the way. In forest management, controlled burning is used in cooler months to prevent devastating forest fires in hotter and dryer times of the year. In the same way, controlled chaos experiments can uncover vulnerabilities that would otherwise cause significant damage.

Several high-cost impact industries and companies have already started practicing Chaos Engineering, including PagerDuty[3] (incident response), the U.S. Air Force[4] (aerospace), ING[5] (finance), and Polysync and Uber[6] (automotive).

Quantitative Benefits: ROI of Chaos

To estimate the ROI for chaos, you need to estimate the *benefits* and *costs* of a Chaos Engineering program. *First-order* benefits and costs are directly observable, and hence the simplest to estimate and reason about.

The main first-order benefit of a Chaos Engineering program is a **reduction in the number of preventable outages**. If the Chaos Engineering program is successful, there should be fewer outages that occur than if the program had not been introduced, all other things being equal.

A Chaos Engineering program has two first-order costs. The first cost is the engineering effort required to implement the program. As an example, Netflix's Chaos Engineering team is made up of four full-time software engineers.

The second cost is the increase in the amount of chaos-induced harm, and the cost of mitigating that harm. Some induced harm will be intentional, and some will be accidental. Intentionally inducing harm may sound odd, but that's part of what Chaos Engineering is about. Every outage preventable by chaos engineering will have corresponding chaos-induced harm that is uncovered by an experiment. The difference is that the chaos-induced harm (if done correctly) should be orders of magnitude smaller than the outage it prevented.

Chaos experiments identify vulnerabilities in the system that would otherwise manifest as outages. By minimizing blast radius during experiments, we work to make the impact as small as

possible while still being detectable. As a result, the impact of chaos-induced outages is not zero. Some outages induced by Chaos Engineering will be, alas, accidental. Any new failure injection program brings with it the risks of accidentally causing problems.

ROI Equation

To calculate the ROI for Chaos Engineering, we need to know:

$C(U)$ – cost of all outages, not using Chaos Engineering
$C(U')$ – cost of all outages, using Chaos Engineering
$C(U'_c)$ – cost of chaos-induced harm

E – effort to implement chaos

$$\text{ROI} = \frac{\text{benefits} - \text{costs}}{\text{benefits}} = \frac{C(U) - C(U') - E}{C(U'_C) + E}$$

Given that:

$$C(U) = C(U_n) + C(U_p)$$

and

$$C(U') = C(U_n) + C(U'_c)$$

where

- $C(U_p)$ – cost of outages preventable by chaos
- $C(U_n)$ – cost of outages not preventable by chaos

The ROI equation then becomes

$$\text{ROI} = \frac{C(U_p) - C(U'_c) - E}{C(U'_c) + E}$$

To do an ROI calculation, we need to estimate:

- the cost of preventable outages, $C(U_p)$
- the cost of harm induced by the Chaos Engineering program, $C(U'_c)$
- the effort to implement the program, E

Estimating the effort to implement the program is straightforward, because you'll choose the staffing levels. Computing the cost of the outages requires some additional data.

Modeling outages using historical data

In an ideal world, we could split the universe into two timelines: in one timeline we institute a Chaos Engineering program, and in another one we don't, then we collect the outage information from each timeline. Unfortunately, we can't do an experiment like that.

To approximate peering into an alternate reality, you can estimate the cost of preventable outages by looking at historical data. If you have a good incident management system in place, you already have a historical record of previous incidents. You'll need to identify which of these could have been prevented if a Chaos Engineering program detected the vulnerability.

At Netflix, our chaos program focuses on the risk of vulnerability to failures in non-critical services (see the sidebar "Surprise! Your Service is Critical"). We look for vulnerabilities where the failure of a non-critical service could put the system in a dangerous state that would lead to an outage.

By identifying which past outages were caused by the failure of a non-critical service, and the impact of these outages, we can estimate how much money we would have saved if we could have avoided them.

Modeling the cost of outages

For the purpose of ROI, once you have a set of historical outages that could have been prevented with chaos, you need to put a dollar value on the cost of the outages.

Depending on your business model, putting a cost on individual outages may be straightforward or difficult. For example, for an online store you can estimate the cost of the outage by the difference between the amount of sales that you would normally process versus the number that were processed during the outage.

We use the term "outage" at Netflix to refer to an incident where some of our customers are unable to stream video. Internally, we assess the impact of an outage by the number of missed streams, the difference between our estimate of how many video streams should be playing (given historical trends) and the observed count of how many video streams are playing.

For Netflix, putting a dollar value on an outage is more complex. Because Netflix uses a subscription model, the relationship between revenue and missed streams are indirect. We believe that an increase in large-scale outages would lead to more people canceling their subscriptions, but coming up with a model that relates outages and cancellation rates is non-trivial. In addition, the effects of outages may be non-linear: one outage that affects 10X users may be more costly to the organization than ten outages that each affect X users, and clusters of outages may be more costly than outages that are evenly distributed throughout the year.

Ultimately, keep in mind that the ROI calculation is based on a simplified model, and it's there to help you make a decision. Your model that estimates costs won't fully capture reality, but the exercise of developing it will help you think more about how outages impact your organization's bottom line.

You'll also need to estimate the cost of the harm that is deliberately induced by Chaos Engineering. For example, if an injected fault intentionally degrades the user experienced (e.g., by serving a non-personalized response), try to quantify the cost the organization of a customer having a degraded experience. This is also a useful exercise because it forces you to make explicit how large of an impact you will allow a chaos experiment to have. You could define an explicit budget for the deliberate impact you will allow the chaos team to have. At Netflix, we automatically detect and stop chaos experiments if they cross a predefined threshold of missed streams. The threshold was chosen to be small enough that it would not trigger existing alerts.

Example scenario

Consider the following (fictional) scenario:

- In the past 12 months there were 2 major outages and 8 minor outages that could have been detected by Chaos Engineering that cost the organization an estimated $700,000 in lost revenue
- A chaos team of three members that cost $150,000 each.
- A budget of $10,000 in missed revenue due to Chaos Engineering experiments

$$\text{ROI} = \frac{\$700,000 - \$10,000 - 3 \times \$150,000}{\$10,000 + 3 \times \$150,000} = 0.5217$$

The estimated ROI would be about 52%.

Qualitative benefits: Influencing Better Engineering Practices

Prof. Daniel Kahneman observed that "no one ever made a decision because of a number – they needed a story."[7] Even if you can demonstrate positive ROI, you'll need to make a *qualitative* argument, tell a good story, to make an effective case for chaos. The second-order benefits of Chaos Engineering, described below, can help build this narrative.

Improving design decisions

Teams which leverage chaos practices while building systems begin to think about failure in a first-class way. Design discussions which used to start with "if component X fails…" become "when component X fails." This is a very subtle, yet powerful, change to strive for because it encourages teams to talk about fallbacks and trade-offs up front and will lead to more hardened and resilient systems in the long run.

When Netflix moved out of the data center and into the cloud, engineers quickly learned about the perilous properties of the cloud: commodity-grade hardware combined with a *scale-out* approach where scale is achieved by adding more servers. The result was that, in the cloud, servers were more likely to disappear, leaving subscribers without service and our engineers scrambling in the middle of the night to restore the service. That's when we created Chaos Monkey. Chaos Monkey randomly terminates production server instances during business hours, when engineers are available to track and fix issues. This quickly uncovered many of our vulnerabilities to instance terminations, making our service resilient to this class of failures. For example, there was a hand-patched "snowflake" server that was responsible for synchronizing some DNS configuration. When Chaos Monkey terminated the server, Amazon replaced it with a server that did not have the hand-patched settings, which resulted in an incident.

More importantly, Chaos Monkey created a cultural shift at Netflix, where engineers now design services to be resilient to instance terminations, knowing that Chaos Monkey could be lurking around the corner.

Understanding service criticality

Because distributed systems frequently suffer from partial failures,[8] it is important to understand which services are the critical ones within your system. For instance, at Netflix the ultimate business goal is to serve video playback to customers. Thus, services that directly impact whether or not playback is available are critical to achieving that goal and we label them Tier 1. For example, a service that returns a digital rights management license, required to decrypt a video, is a Tier 1 service. However, a service which provides a personalized recommended list of movies is not critical to video playback, and we'll refer to such services as Tier 2.

This distinction provides a useful mechanism for defining service tiers (see sidebar: Surprise! Your service is critical). Netflix Chaos Engineering focuses on two tiers, but you could define as many as you need for your testing.

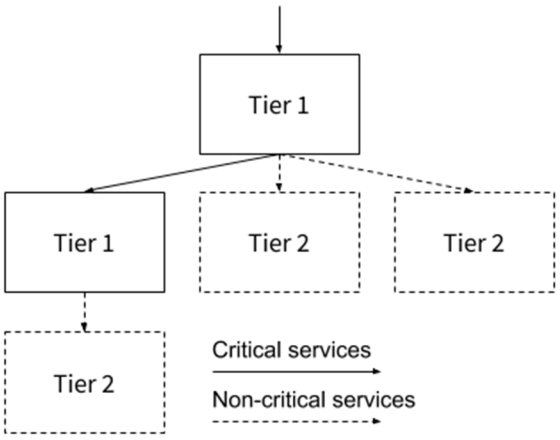


Figure 1. Tier 1 and Tier 2 services

Teams which own services should (1) know which type of service they are operating and (2) be able to demonstrate that they are not more critical than their designated tier. Chaos experiments enable teams to do this in a contained manner.

Assuming each tier has different requirements for on-call and production-readiness, this can also be a motivating factor in prioritizing work on resiliency efforts. For example, teams may choose to spend additional engineering resources to build in fallbacks if it allows them to move their service into a lower tier with less stringent operational requirements.

Validating reliability measures

There are many tools which can be leveraged to make distributed systems resilient to system faults. A few examples of scalability patterns include timeouts, fallbacks, circuit breakers, and bulkheads.[9] Like all software, these mechanisms can be independently tested in the small via unit tests to ensure they function as desired. However, the true test of these strategies is to see how they react in a production environment under load, particularly when you consider that each one requires tuning and configuration to set boundary conditions.

One example of a reliability measure we use at Netflix is Hystrix. Hystrix implements the circuit breaker pattern for wrapping RPC calls. This enables teams to define a fallback that is served in the case of a service outage. If the Hystrix error percentage crosses a configured threshold, the circuit opens and serves a fallback instead of making calls to downstream services. It then slowly retries the RPC calls at a lower rate until the failing service recovers. This is a useful tool for protecting the calling service from backing up threads, but it only works if the error threshold is configured correctly and the fallback functions as expected. We have seen outages as a result of failures in this code and configuration, so in order to ensure Hystrix barriers continue to function, we are increasing chaos experiments to validate them on a regular, on-going basis.

AFTER BUY-IN: BEGINNING THE CHAOS

The Role of Chaos in Organizations

Once you've received approval from your organization to start a chaos program, the first step is to specify the success criteria for your business. Three items should be analyzed prior to beginning chaos experimentation: business goals, key performance indicators (KPIs), and the cost of impact to service availability.

Acquiring and retaining customers are ubiquitous business goals across many industries; both of which are severely impacted if functionality of your core product is unavailable.

To measure progress against business goals, engineering teams monitor key performance indicators (KPIs), or metrics, which directly impact the success of the business. Technology teams and business partners should align on KPIs before experimentation begins. For example:

- For a home security company, a KPI may be the time between a security panel being armed and doors being locked.
- For an ecommerce company, this might be the number of customers checking out, searching for an item, or adding an item to a shopping basket.
- For a ride-sharing company, it may be the volume of rides hailed in a given minute.
- For a video streaming company, this might be how many streams are started in a given second.

In order to monitor Chaos experiments, a baseline for these metrics should be established which can be based on historical data. The baseline is then compared to the experiment KPIs to understand whether an experiment is causing impact.

The cost of impact varies depending on your business; it could be the cost of acquiring a customer, a safety impact, the engineering hours spent resolving the issue, phone calls to your Customer Service team, or a combination of the aforementioned.

The relationship between business goals, cost of impact, and KPIs will give you a strong indication of whether Chaos Engineering can, and should, be applied to your organization. A high cost of impact, associated with a given KPI, associated with a key business goal should be regularly experimented on through Chaos Engineering to ensure a higher likelihood of success. This can be done by monitoring KPIs while failing various components that fuel them in order for engineers to have a better understanding of the effect and potential blast radius[1] of a failure.

Running Chaos Experiments

Ideally, you would run chaos experiments any time your system changes, just as you would with any other type of testing. However; there is risk of impacting customers with these experiments, so there is a balance to strike between minimizing impact to customers and finding the vulnerabilities quickly before it can turn into a larger problem. This becomes more difficult the more complex your environment is. For instance, if you have mechanisms for people to dynamically push configuration updates into the environment, that is a change to the system.

At Netflix, there are tools like Chaos Monkey that run all the time in production and they have become part of the engineering culture. As we dive into more large-scale types of testing, we are more cautious, yet the frequency has increased over the last few years. Originally, there was a push to run experiments manually once or twice leading up to a major release or holiday code freeze. This enabled teams to bolster their systems for that particular event; however, as soon as we resumed changes the systems were vulnerable. As a result, we are moving toward more automation for these experiments. Many service owners are now running chaos experiments weekly or monthly depending on the risk associated with running those experiments. As the tooling improves to catch problems (and kill bad experiments quickly), more experiments will be possible. For more details on how to design and run Chaos Engineering experiments, see the work of Rosenthal and colleagues.[10]

Adoption Evolution

The true mark of success for introducing new technology into an organization is whether or not it changes the process, and if that process change is for the better. While it's important to monitor the quantitative aspects of Chaos Engineering, it's equally important to measure qualitative aspects. Throughout the adoption process, measure your successes and how things are changing within the organization.

Different cultures and industries approach Chaos Engineering adoption a bit differently. You likely aren't going to be able to convince everyone of its potential success immediately. We suggest starting with a critical or "Tier 1" service. Getting a critical service to experiment with Chaos Engineering is useful in many ways:

- Downstream services are more likely to adopt if a leader service adopts first
- Higher ROI with Chaos Engineering
- Less time spent mitigating outages

For your first pass, start with a previous outage and go backwards. Before beginning chaos, you should be able to answer the following questions about the outage you're recreating: which services were impacted, what calls failed, and where the issue started from.

Most service owners won't object to continuous experimentation on something that has the potential to be a high-cost outage to the organization in order to prevent it from happening again. Once this is in place, gradually expand the chaos experiments towards other services that behave similarly and continue doing this until you have full "basic" adoption. After basic adoption is achieved, you can increase granularity with your experiments in terms of latency and failure based scenarios.

As your adoption evolves, Chaos Engineering should become more automated. However, it's recommended to start with a consultancy model when introducing chaos in order for service owners and business owners to be kept in the loop with regards to potential risk and reward. Once you feel comfortable teams have an understanding and involvement in the Chaos Engineering practice, begin to remove yourself from the equation and allow the tool to handle more and more of the responsibilities.

CONCLUSION

Because Chaos Engineering is still a young discipline, attempts to introduce a chaos program are likely to be met with skepticism. By telling a story around the benefits of chaos realized at organizations such as Netflix, and by making quantitative arguments based on your historical data, you should be able to make a strong case that introducing Chaos Engineering will yield real, tangible benefits to the business.

SIDEBAR: HOW CHAOS RELATES TO TRADITIONAL TESTING

To understand the goals of Chaos Engineering, it is important to see where it fits into the larger development life cycle.

Unit tests are critical for validating expected results at a very granular level and unit tests often mock dependencies to run as consistently and quickly as possible. The goal of unit testing is to validate what an *individual component* is doing—in other words, given certain inputs, determine the expected outputs.

Integration tests have a similar goal – however, they attempt to verify that components work together to achieve the desired result. As microservice architectures have grown more popular, integration testing encompasses not only the interaction of components within a single service, but also across services.

While integration and unit testing are valuable for identifying defects, they don't cover everything. In addition to ensuring a system is functionally correct, chaos experiments validate assumptions about how *systems* react to different types of faults between components. Distributed systems fail often. They also fail differently under production load than they do by running a single failure scenario as part of a unit or integration test. By showing how each component within the system behaves under failure conditions, Chaos Engineering exposes system vulnerabilities.

This discipline enables platform and service owners to build confidence in resilience mechanisms such as retries, timeouts, circuit breakers, load balancers, and back pressure.

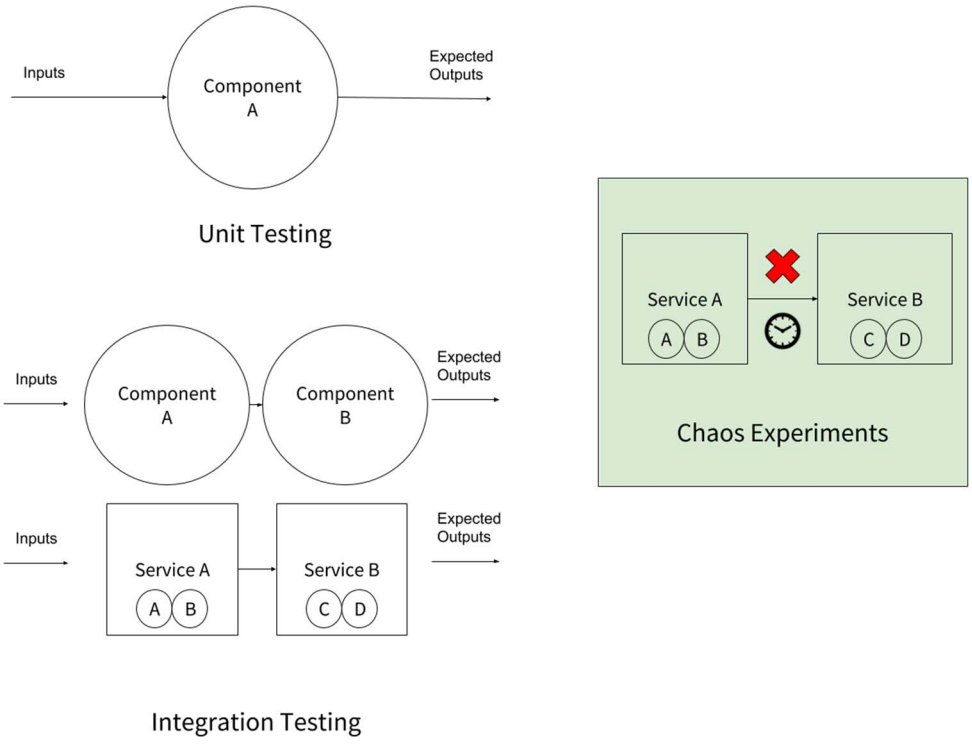


Figure 2. Unit Testing, Integration Testing, and Chaos Experiments

A single service may have thousands of unit and integration tests validating expectations as defined by the business, but without chaos experiments, it's impossible to know that the system as a whole will be fault tolerant and available for your customers.

SIDEBAR: SURPRISE! YOUR SERVICE IS CRITICAL

In a complex distributed system, a microservice architecture can easily become a distributed monolith, where a fault in any part of the system can take down the entire system. At Netflix, we use fallbacks and circuit breakers to protect our critical services from faults in non-critical systems. We define a critical service to be any service that is required for selecting a video and playing it back. For example, we have a non-critical service that tracks where to resume a movie for a customer that left halfway through. If that service goes down, the fallback would be to play from the beginning of the movie during the downtime. Another service displays a personalized list of movies and shows to users based on their past viewing preferences. If that service goes down, the fallback would be to display a list of top movies and shows to our customers. Sometimes our non-critical services accidentally become critical. In order to find these types of problems faster, we've built a tool called Chaos Automation Platform (ChAP) to inject failure and latency into non-critical services in production in order to observe their impact on critical services in a contained manner.[11]

REFERENCES

1. *Principles of Chaos Engineering*, April 2017; http://principlesofchaos.org/.
2. N. Popper, "Knight Capital Says Trading Glitch Cost It $440 Million," *New York Times DealBook*, 2 August 2017; https://dealbook.nytimes.com/2012/08/02/knight-capital-says-trading-mishap-cost-it-440-million.
3. K. Rose, "Failure Friday: How We Ensure PagerDuty is Always Reliable," *PagerDuty*, blog, 20 November 2013; www.pagerduty.com/blog/failure-friday-at-pagerduty.
4. J. Cation, "Flight Control Breakthrough Could Lead to Safer Air Travel," *Engineering at Illinois*, Illinois College of Engineering, 19 March 2015; https://engineering.illinois.edu/news/article/10817.
5. B. Wong et al., "Chaos Engineering: A Lesson From the Experts," *Performance Zone*, DZone, 29 July 2017; https://dzone.com/articles/chaos-engineering-a-lesson-from-the-experts.
6. "Chaos Community Day," September 2017; https://chaos.community/.
7. M. Lewis, *The Undoing Project: A Friendship That Changed Our Minds*, W.W. Norton & Company, 2016.
8. J. Hodges, "Notes on Distributed Systems for Young Bloods," *Something Similar*, blog, 14 January 2013; www.somethingsimilar.com/2013/01/14/notes-on-distributed-systems-for-young-bloods.
9. M. Nygard, *Release It! Design and Deploy Production-Ready Software*, Pragmatic Bookshelf, 2007.
10. C. Rosenthal et al., *Chaos Engineering: Building confidence in system behavior through experiments*, O'Reilly Media, 2017.
11. A. Basiri et al., "A Platform for Automating Chaos Experiments," *2016 IEEE International Symposium on Software Reliability Engineering* (ISSRE 16), 2016, pp. 5–8.

ABOUT THE AUTHORS

Haley Tucker is a senior software engineer on the Chaos Engineering team at Netflix, where she is responsible for verifying the resiliency of Netflix services to ensure that customers always enjoy their favorite shows. Tucker received a BS in Computer Science from Texas A&M University. Contact her at haley@netflix.com.

Lorin Hochstein is a senior software engineer on the Chaos Team at Netflix, where he works on ensuring that Netflix remains available in the face of continual production changes. Hochstein received a PhD in Computer Science from the University of Maryland. Contact him at lhochstein@netflix.com.

Nora Jones is a senior software engineer on the Chaos Engineering team at Netflix, where she works on ensuring that Netflix remains resilient in the face of uncertain conditions. Jones received a BS in Computer Engineering from Virginia Tech and is a currently pursuing an advanced degree in Human Factors and Systems Safety at Lund University. Contact her at noraj@netflix.com.

Ali Basiri is the founder of the Chaos Engineering team at Netflix where, he works on ensuring Netflix's availability through a series of resiliency initiatives such as Chaos Automation Platform (ChAP), Chaos Monkey, and FIT. Basiri has a BMath in Computer Science from the University of Waterloo, Canada. Contact him at abasiri@netflix.com.

Casey Rosenthal formalized the practice of Chaos Engineering by co-writing and publishing the definition principlesofchaos.org with the Chaos Team at Netflix, which he managed for three years. Rosenthal is currently CTO at Backplane.io, a company that provides reliable and resilient infrastructure. Contact him at casey@backplane.io.

A Virtual-Machine Resiliency Management System for the Cloud

Valentina Salapura
IBM Watson Health

Rick Harper
IBM Watson Health

This article presents a scalable parallel virtual machine (VM) resiliency management system for the cloud environment. It describes two complementary mechanisms for providing resiliency: automated VM restart when a physical server fails unpredictably, and automated evacuation of the VMs from a physical server that is about to fail or needs to be maintained. The solution is suitable for clouds with a large number of physical servers, VMs, disks, networking components, and management tools. This solution has been deployed in IBM's Cloud Managed Services enterprise cloud.

Critical workloads in the cloud must be resilient to a variety of potential threats to dependability. Chief among these threats are unplanned outages due to unforeseeable hardware failures and planned outages due to system maintenance and proactive repair.

Certain problems arise from the large scale of the system and other constraints to be described below. A potentially large number of virtual machines (VMs) must be either restarted or evacuated, especially in scenarios where multiple physical servers are affected by the event and only a short time window is available to complete the resiliency processes. Every step of the resiliency process must be designed to be either prestaged (as in creation of a precomputed failover plan), performed very quickly (as in, alternatively, very fast failover plan computation), or performed in parallel (as in parallel disk mapping and VM migration).

To provide resiliency against an unplanned failure of one or more physical servers, it is necessary to automatically detect that one or more servers have failed, compute a failover plan for distributing the affected VMs among the surviving servers, and restart the affected VMs on alternate physical servers. In this scenario, the impact to the affected VMs is as if the VMs had crashed and been restarted. Once the failed server has been repaired, the failed-over VMs are rehomed to their original physical servers using Live Partition Mobility (LPM, also known as Active Migration),[1] which migrates the VMs from one server to another while they are still running. During the LPM procedure, there is a subsecond interruption to the workload, which is usually

Copublished by the IEEE CS and IEEE ComSoc
2325-6095/18/$33.00 USD ©2018 IEEE

unnoticeable. If the physical server cannot be repaired or is to be decommissioned, the failed-over VMs remain on their failover target physical servers.

To provide resiliency against planned outages of one or more physical servers, the resiliency management system computes an evacuation plan that distributes the affected VMs from the server or servers to be maintained to the unaffected servers. It then automatically evacuates these VMs in parallel, using LPM. After the planned operations have been completed and the serviced physical servers have been brought back online, the resiliency management system then rehomes the migrated VMs back to their original physical servers. If some of the original physical servers are not or cannot be brought back online, the migrated VMs remain on the servers to which they were evacuated.

Both of the above-mentioned resiliency management functions share similar functionality. Failover and evacuation target planning is a common function that has to be invoked regardless of whether it is a response to a physical failure or it is preparatory to a planned evacuation. Both failover and LPM require that disks be dynamically mapped to the failover or evacuation target. Both failover and evacuation require the ability to rehome the affected VMs. Finally, to minimize the time required for failover or the duration of a planned-maintenance window, it is crucially necessary to squeeze the maximum possible parallelism from the failover, disk-mapping, VM restart, and VM LPM operations.

This article describes the implementation of components of the resilience management system in IBM's Cloud Managed Services (CMS) enterprise cloud offering, which handles both unplanned physical-server outages and planned physical-server maintenance, while minimizing the disruption to the workload due to these phenomena.

This article has three main contributions. First, it illustrates the challenges of a combined resiliency management system that handles both unplanned and planned outages of a physical server supporting a virtual workload. Second, it highlights commonalities between handling these two impairments to dependability. Finally, it describes the modular design and performance of such a system in the context of IBM's CMS environment.

THE CMS POD ARCHITECTURE

CMS (http://www.ibm.com/marketplace/cloud/managed-cloud/us/en-us) is an IBM cloud-computing offering for enterprise customers.[2] It is designed to bring the advantages of cloud computing to the strategic outsourcing customers of IBM. It provides standardized, resilient, and secure IBM infrastructure, tools, and services with full ITIL (Information Technology Infrastructure Library) management capabilities. CMS also offers functions such as consumption-based metrics and automated service-management integration.

The design of CMS is based on a unit called the *point of delivery* (PoD). PoDs are cloud instances that are deployed and managed individually and independently. A PoD contains many physical managed resources (servers, storage, and the network) that are virtualized and are provided to customers as an infrastructure offering. A CMS PoD contains Intel-based servers to support virtual and bare-metal Windows and Linux workloads, and IBM Power servers to support virtual AIX workloads.

A PoD is designed to be highly available, with the physical infrastructure architected to eliminate single points of failure. On top of this infrastructure, availability management software and other functionality have been implemented that allow CMS to offer customers selectable service-level agreements (SLAs), which are contractual obligations and may include penalties for noncompliance. The availability SLAs pertain only to unplanned outages and apply solely to VM availability.[3] CMS supports multiple levels of availability, ranging from 0.985 to 0.999.[2] In practice, on the basis of the specified availability SLA, a maximum-allowable monthly downtime budget is specified. VM monitoring tools measure the uptime, and if the allowable downtime is exceeded within a given month, that constitutes an SLA violation.

Management tools for storage management, backup, and performance monitoring are hosted on managing servers, which are also part of a PoD. Managing servers host tools for controlling, provisioning, managing, and monitoring the workload on the managed servers. Relevant managing

tools for this discussion are Tivoli Service Automation Manager (TSAM), which maintains the PoD management information such as SLAs, Tivoli Storage Productivity Center (TPC) for storage management, and the Hardware Maintenance Console (HMC)[4] for server management.

APPROACHES TO HIGH AVAILABILITY IN A VIRTUAL ENVIRONMENT

Enterprise-class customers, such as banks, insurance companies, or airlines, typically require IT management services such as monitoring, patching, backup, change control, high availability, and disaster recovery to support systems running complex applications with stringent IT process control and quality-of-service requirements. Such features are typically offered by IT service providers in strategic-outsourcing engagements, a business model for which the provider takes over several or all aspects of management of a customer's datacenter resources, software assets, and processes. Servers with such support are characterized as being managed.

This is in contrast to unmanaged servers provisioned using basic Amazon Web Services (AWS)[5] and IBM's SoftLayer offerings. In these offerings, the cloud provider offers automated server provisioning and enables the user to add services to the provisioned server, but assumes no responsibility for their upkeep. AWS provides on-demand IT resources via the Internet with a pay-as-you-go pricing model.

Within the continuum of these approaches to the degree of management provided by the cloud offering, high availability (HA) in a virtual environment can be achieved in multiple ways. One possibility is to implement HA at the infrastructure level, as we describe in this article. Another approach is to provide HA at the application level, by using HA clustering techniques. A different approach is to design an application to tolerate the loss of one or more VMs, which is how many "born in the cloud" applications are designed. While in this article we address the infrastructure-level HA techniques in detail, other HA approaches are also implemented and supported in the CMS cloud. We have found that infrastructure-level HA and application-level HA can be designed to operate beneficially together with no mutually destructive effects.

HA solutions at the infrastructure level are designed to ensure that the virtual resources meet their availability targets. This is accomplished by continuously monitoring the infrastructure environment, detecting a failure, and invoking recovery procedures when a failure occurs. Typically, such recovery procedures involve restarting the failed VM, on either the same or a different physical server.

A widely used product for infrastructure-level VM management, monitoring, and recovery is VMware HA.[6] This solution has the ability to monitor virtual and physical resources and perform automatic-recovery actions. It, however, requires predefining a set of servers and storage that these servers can share. Workload migration can be enabled only within such a defined set, limiting the flexibility of the solution and leading to suboptimal resource usage within the larger cloud instance. For example, a workload can migrate only within these servers, even if they're being heavily used and other physical servers outside the enabled set are being underutilized. Recently, more IaaS (infrastructure as a service) cloud providers have started to focus on increasing cloud infrastructure availability—a good review is provided in "Thinking about Availability in Large Service Infrastructures."[7]

Application-level HA techniques are built around application-aware clustering technology. These solutions are used to improve the availability of applications by continuously monitoring the application's specific resources and their physical-server environment and invoking recovery procedures when failures occur. These solutions typically use multiple VMs that are working together in order to ensure that an application is always available. These VMs are arranged in an active–passive or active–active configuration. When one VM fails, its functionality is taken over by the backup VM in the cluster. Examples of these solutions are IBM PowerHA, Microsoft Clustering Services, Symantec Cluster Server, and Linux-HA.

Another approach to building highly available applications is to design them around cloud-computing principles. Such applications are typically highly parallel, stateless, shared-nothing (as opposed to shared storage) applications employing multiple VMs. The state of the application is

typically eliminated. If that is not possible, the state is minimized and pushed to a database on secure storage, which itself would require infrastructure-level resiliency. In this class of applications, loss of any VM does not impact the overall application, other than perhaps a performance degradation.

CMS fully supports the application-level HA methods described above. However, because this article focuses on infrastructure-level resiliency, these techniques are not discussed here.

ARCHITECTURE OF THE VM RESILIENCY MANAGEMENT SYSTEM

We designed the resilience management system in the cloud to be modular, extendable, and versatile. The modular system comprises several different units developed over several years, each accomplishing its specific task. Both the systems for handling unplanned system failover and for handling planned maintenance require certain identical functions. Such functions include

- destination planning for the VMs that are to be migrated on the basis of the available resources in a PoD and
- dynamic storage and network configuration for these VMs on their destination servers.

Additionally, there are functions specific only to failover or to planned maintenance.

The modules of the VM resiliency management system are illustrated in Figure 1. The middle of the figure illustrates tasks that are common to both recovery from unplanned system failure and planned-maintenance tasks. These modules are configuration data collection, DynaPlanner, disk management, and network configuration. The modules on the left—server status monitoring, restart priority determination, and logical partition (LPAR) restart—are specifically for handling unplanned failover. Finally, the tasks on the right—maintenance setup and LPM—are for planned maintenance only.

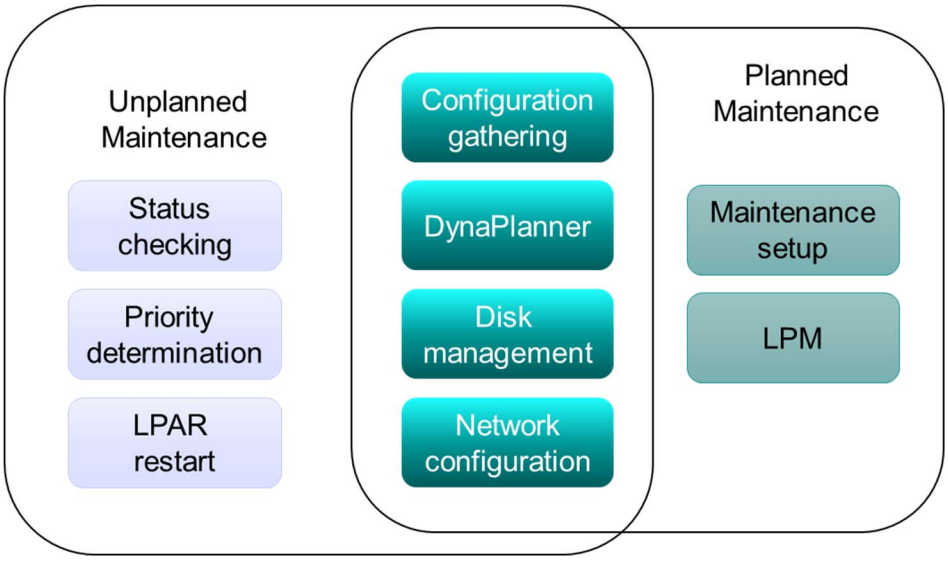


Figure 1. The modules of the virtual machine (VM) resiliency management system, unique to failover (unplanned maintenance), unique to planned maintenance, and shared by both. LPAR = logical partition, and LPM = Live Partition Mobility.

The IBM Power VMs are called LPARs. The configuration-data-gathering-and-persistence unit collects the configuration and status of LPARs in a PoD. For unplanned-failover handling, these data are collected periodically, and the time interval for data gathering is configurable. For

planned maintenance, the configuration data are collected at the beginning of maintenance. Information about all the physical servers in the PoD, the hosted LPARs, and the connected storage is collected via an HMC. The SLA availability information for all LPARs is obtained by querying the TSAM database.

We refer to the system for handling system failover as *remote restart*, and its operation is as follows. The server failure detection unit monitors the health of all servers in a PoD by polling the HMC for the status of each physical server. A failure of a server is indicated by the HMC when the poll returns an ERROR state. The polling interval is configurable and is set to a value that allows the SLAs to be met. When a server failure has been confirmed, the failover-planning task uses the DynaPlan[8] algorithm to determine the optimal failover targets. The algorithm determines the targets on the basis of the resources available in the PoD. Once this has been done, the failover tasks themselves can proceed with server fencing, network and storage dynamic configuration, and restarting the LPARs. The server-fencing unit powers off a server via HMC commands, once a server is determined to be faulty.

Planned maintenance uses the same failover-planning mechanism and the same data collection mechanisms. It starts these mechanisms at the beginning of the maintenance phase to determine for each VM hosted on the maintained server where to migrate it on the basis of the resource utilization in the PoD.

Both solutions use the network configuration and storage management modules. The storage disk management unit creates virtual SCSI (Small Computer System Interface) logical disks (denoted by their Logical Unit Number, or LUN) on the failover destination servers. It then maps those LUNS to the LPARs to be restarted using the APIs and the functionality of the HMC. Since all LUNs are not usually connected to all servers in a PoD, the storage disks are connected dynamically to the failover targets according to the evacuation plan, via TPC commands. For remote restart, once a SCSI and LUNs are created and connected to the failover server, each LPAR is created and restarted on the failover server via HMC commands.

The restart priority of LPARs for remote restart is based on the LPARs' SLAs. LPARs with an availability SLA of 0.999 are restarted first, followed by the LPARs belonging to lower-SLA groups. For planned maintenance, SLA information to determine the migration priority is not used. Instead, live migration is initiated for LPARs hosted on the server that is being maintained, to move them to their destination servers.

Virtualized infrastructures can be designed such that all physical servers in a server pool are statically mapped to all the storage devices that may be used by the VMs. However, we opt to dynamically map physical machines only to the storage devices that are needed to support the virtual workload running on the respective physical machines. The reason for this decision is that the static-mapping approach is unsuitable for large-scale cloud environments, where the pool may consist of hundreds or more servers, supporting thousands of VMs, which in turn use tens of thousands of storage devices. The architectural and design limits of the hypervisors cannot support the huge number of simultaneous connections required to support all possible VM-storage device mappings. Instead, we opt to map a physical server to storage for only those VMs that are running on that physical server. Because of its large scale, the CMS PoD adopts the dynamic-mapping approach for the IBM Power server subsystem.

Planned Maintenance

It is occasionally necessary to power down a physical server to perform planned or emergency maintenance. Such maintenance may be required for firmware upgrades, to increase the capacity of the physical server, to replace redundant physical components, to replace or remove the server itself, and to deal with other rare events. To prevent this server outage from impacting the availability of the LPARs running on the impacted server, it is necessary to evacuate those LPARs to other servers using LPM.

LPM is an IBM Power server function whereby a running LPAR is moved to another physical server with minimal interruption of its operation. It is equivalent to VMware vMotion and other approaches well known in the industry, although the mechanisms differ significantly for the IBM Power platform.

The architecture for performing evacuation in support of planned maintenance is similar to the remote-restart architecture, and many of the functions are reused. Data gathering, the failover planner, disk mapping and unmapping, and HMC access are all used. An additional script is added alongside the remote-restart script that performs the orchestration of the evacuation and rehoming when invoked by the steady-state service personnel. As in remote restart, parallelization of certain operations is required in order to complete the evacuation process within the CMS mandated maintenance window. All Power servers in the pool must be connected by at least a 10 Gbit/s network to perform the memory copying, in order to allow the evacuation to proceed within the maintenance window. Note that this network is not needed by remote restart because, unlike LPM, it does not copy the LPAR's memory contents from one physical server to another.

When planned maintenance is required, an evacuation plan is first created. For each destination, a physical server is identified as an evacuation target, the LUNs used by the migrating LPAR are mapped to that physical server, and the LPAR is migrated to that server using commands to the HMC. The HMC creates an LPAR, configures the Virtual I/O Server at the destination server, and copies the memory of the source LPAR to the destination server using a convergent memory copy algorithm. When the copy is complete, the LPAR on the primary server is stopped and the LPAR on the destination server is started.

During the final phase of LPM, the LPAR experiences a pause of approximately one second or less. Certain highly time-sensitive applications are sensitive to this tiny outage and should be quiesced prior to LPM. One example is an LPAR that is a member of an IBM PowerHA cluster. The subsecond interruption of the LPAR has been known to cause a PowerHA "deadman timer" to fire, thus triggering a PowerHA failover. This is avoided by quiescing this timer prior to the LPM process and reenabling it after LPM is complete. Other LPARs cannot be moved owing to nonvirtualization licensing models that pin the LPAR to a given physical server, or owing to the requirement that they run only on a physical server that contains a particular hardware capability. In these cases, it is necessary to create an exclusion list that notifies the evacuation procedure to not attempt to migrate these LPARs.

After the evacuation of a physical server has been completed, the necessary maintenance of that server can be performed. Once the maintenance has been completed, it is usually desired to move the evacuated LPARs from the physical servers to which they were evacuated, back to the original server they resided on prior to the maintenance. This rehoming of the LPARs is accomplished by an LPM process similar to evacuation.

Recovery from Unplanned Failures

Although the IBM Power servers used in CMS contain extensive internal redundancy and are capable of predicting their own failures and triggering remediation action, it is still possible that a server could suffer an unanticipated failure. In this failure mode, the server suffers a hardware failure from which it cannot recover in a short time (for example, 10 minutes) and for which it requires maintenance or repair. In this case, the failover process will restart all hosted VMs on another server in the same PoD. The recovered VMs need to use the same storage disks—the LUNs—that the original server was using.

We recover VMs upon failure of a physical server by restarting the affected VMs on alternate physical servers, similar to VMware.[6] Alternate VM resiliency techniques perform synchronous replication from a primary VM to a secondary VM with very low recovery times.[9] However, these were not selected because of the high network bandwidth utilization consumed by the replication, the difficulty in supporting multiprocessors that is endemic to these techniques, and the difficulty of supporting replication at a distance. Furthermore, the CMS SLAs did not require a fault-tolerant solution, and a restart-based solution was determined to be able to meet the SLAs.

Restart priority is a partial ordering of all VMs into priority classes, subject to all the considerations mentioned above. Within a given priority class, all VMs can be restarted in parallel, subject to the restart parallelism constraints of the physical environment and application-start-order dependencies.

The restart priority is automatically and dynamically determined on the basis of a number of VM properties, such as the SLA, application priority, application topology, and other rules as determined by the dynamic restart priority calculator and a given set of rules. Priority aggregation rules convert the various restart rules into the VM restart partial priority order, while taking into account application dependencies.

Figure 2 illustrates the grouping of VMs into different groups based on the number of target servers available. The figure gives an example where four destination servers are available. In a PoD, the number of destination servers is higher, increasing the restart parallelism. For each target server, a restart queue is formed, where each VM is assigned to only one queue. All VMs are restarted on the target servers on the basis of their restart priority. For illustration purposes, we use three restart priority classes: H (high), M (medium), and L (low). In reality, the CMS cloud uses four different priority classes. For each destination server, VMs from different priority classes are assigned. All restart queues are processed in parallel.

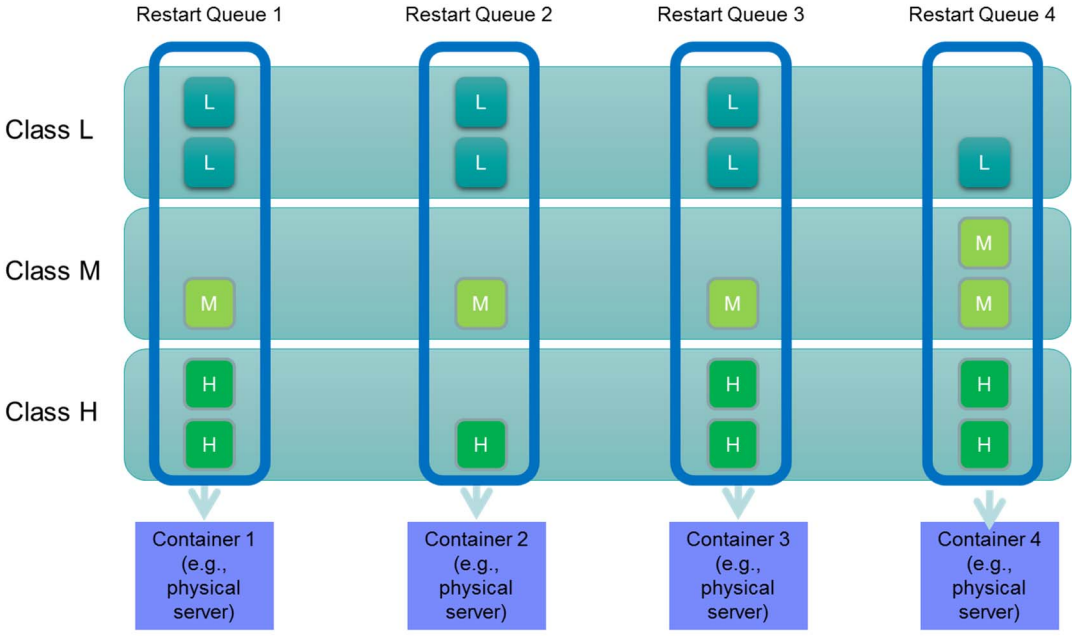


Figure 2: Grouping VMs for restart on different target servers. The restart within a group is determined on the basis of the service-level agreement. L = low, M = medium, and H = high.

Within each restart queue, the VMs with highest priority are restarted first. The restart queues operate in parallel and independently of each other, and it is possible that the VM restart per priority class gets out of absolute order. For example, for the target Server 2, if a low-priority VM has a few disks attached, it could restart simultaneously with a high- or medium-priority VM scheduled for the target Server 4. Although this could happen, each individual class is guaranteed to complete in its allowed time.

While this parallelization allows all SLAs to be met for most configurations of the PoD, it was found that for the worst case, restart parallelism is insufficient to meet the SLAs. This is the case for a half PoD with a small number of servers and for fully loaded servers hosting the maximum number of LPARs with a large number of LUNs. With the smallest-available restart parallelism, the time needed for failover for a large number of LUNs cannot be guaranteed to fit within all the SLAs' time budgets.

Thus, our next improvement focused on parallel disk mapping. In this implementation, in addition to starting parallel failover processes, we also perform the mapping of multiple disks attached to a single LPAR in parallel. This method ensures recovery within the time allowed by the SLA, even for the worst-case scenario.[10,11]

Considerations for Failover and Evacuation Planning

The VM resiliency management system uses the DynaPlan[8] resource planner to formulate a plan to evacuate or restart a large collection of interdependent VMs on a large collection of resources. DynaPlan is a placement algorithm that inputs

- the VMs' multidimensional resource requirements (such as CPU, memory, and other consumable resources);
- the equivalent capacity metrics for the environment;
- various constraints such as collocation, anticollocation, and licensing; and
- whether the placement is to be energy-optimizing, performance-optimizing, or somewhere in between.

On the basis of these inputs, DynaPlan creates a map that distributes the VMs across the environment while satisfying all constraints. The detailed operation of DynaPlan in this application is not covered in this article—we focus instead on certain factors and constraints arising from the characteristics and capabilities of the PoD's resources and on the organization of these resources into fault domains. Additionally, for the purposes of failover planning, it is necessary to construct an optimal parallel-restart plan that does not exceed the recovery bandwidth (i.e., the number of parallel recovery operations allowed by the infrastructure) yet meets SLAs (i.e., the maximum allowable recovery time).

PoD Resource Characteristics

It is often assumed that all the servers in a server pool are homogeneous, so that an LPAR can move from one server to an identical one. However, this is difficult to ensure in the cloud, as servers are often replaced with better, faster models. Thus, if a server pool is heterogeneous and the target server is different from that of the source, then the LPAR's CPU configuration has to be adjusted so that the net computational capacity delivered to that LPAR, as measured by some industry benchmark, meets that LPAR's performance requirement.

Much large enterprise software, such as databases, is licensed by cores, and the licenses' restrictions demand that the software be installed on the same physical server in the cloud for its entire lifecycle. This is an unfortunate remnant of the precloud nonvirtual-datacenter model. Many of these license restrictions remain in effect in the cloud, and failover planning must take this into consideration. The choices are to either save on licensing costs and not have the ability to move these LPARs, or purchase additional contingency licenses so that it is possible to move them to the specific servers for which they are licensed.

Virtualized storage architectures, in which disks are mapped from the storage appliance through a storage area network (SAN) virtualization layer to the server and then to the LPARs, can impose restrictions on failovers. The general load-balancing rule is to maximize performance by distributing the disks across different I/O groups in the SAN virtualization layer. When an application retrieves data from, or writes data to, different disks, it will use different I/O groups and is therefore less likely to face I/O bottlenecks. Failover planning must survey the I/O group distribution of the already existing LPARs on the target server alongside the newly migrated ones and make sure that there is still adequate I/O distribution for all the LPARs and their disks.

Storage tiering adds another dimension to dealing with I/O performance. Storage (disks) from different tiers have different I/O performance characteristics. In addition to disks being balanced across I/O groups, they have to be balanced by net I/O demand.

Physical Fault Domain Structures

LPARs are often provisioned in multiples to support application-based high availability as described above. This means each member of this high-availability cluster is placed, during initial provisioning, on a different physical server to isolate against server outages, planned or unplanned. The planner should observe this restriction during any evacuation. Generally, this is expressed as "should" rather than "must" because of the number of LPARs that must be placed

versus the available physical servers for placement. This is because it is usually deemed more important to restart the LPAR anywhere than to temporarily violate anticollocation constraints for the duration of an outage or a maintenance activity.

Physical servers in a cloud are placed on racks, and the racks are placed into rows. These racks have one or more power sources. Initial placement of LPARs should be controlled to avoid placing clustered pairs within the same power domain. The planner has to consider this when migrating LPARs. Unless there is a plan to rehome the LPARs, the target servers should also span different power domains.

CMS PoDs are built in logical halves that can be physically separated and placed in datacenters miles apart and connected by dark fiber. This type of separation is called a "dual-room" PoD. This allows fault domains to withstand localized outages due to power failures, floods, building damage, etc. The planner's placement policies are required to take such physical constructs into consideration. When a catastrophe strikes one "room," then there is no choice but to place all the LPARs in the alternate room.

Restart Performance Constraints

The planner has to meet several conflicting performance constraints, such as that

- all recovery time objectives are met,
- the maximum number of the most important dependency groups is started,
- VMs within a dependency group are started in the correct order, and
- the capabilities of the environment (e.g., restart bandwidth and capacities) are not exceeded.

The failover planner is run at the beginning of maintenance or at failure-handling time, since the cost of running is extremely low. In addition, the failover planner is run once per day for each server in a PoD to determine if there are any capacity problems that prevent all LPARs on that server from failing over to the remaining hosts. If this condition is detected, a warning notification is sent to the cloud administrators for the purposes of remediation.

CONCLUSION

In this article, we have presented a VM resiliency management system. We have described two mechanisms for providing resiliency: automated VM restart when the hosting physical server fails unpredictably, and automated evacuation of VMs from a physical server that is about to fail or needs to be maintained. The automated VM restart system includes failover-time planning and execution and keeps the recovery time within the budget allowable by the applicable SLAs. The work that was required to achieve this performance at scale is also described in this article. This restart implementation forms the basis for the automated evacuation of the VMs from a physical server.

REFERENCES

1. *Live Partition Mobility PDF*; www.ibm.com/support/knowledgecenter/8412-EAD/p7hc3/abstract_lpm_guide.htm.
2. A. Kochut et al., "Evolution of the IBM cloud: Enabling an enterprise cloud services ecosystem," *IBM Journal of Research and Development*, vol. 55, no. 6, 2011, pp. 397–409.
3. V. Salapura, R. Harper, and M. Viswanathan, "Resilient cloud computing," *IBM Journal of Research and Development*, vol. 57, no. 5, 2013, p. 10:1.
4. *Hardware Management Console*; www.ibm.com/developerworks/community/wikis/home?lang=en#!/wiki/Power+Systems/page/Hardware+Management+Console.
5. F.P. Miller, A.F. Vandome, and J. McBrewster, *Amazon Web Services*, Alpha Press, 2010.

6. *VMware High Availability*; www.vmware.com/files/pdf/VMware-High-Availability-DS-EN.pdf.
7. J.C. Mogul, R. Isaacs, and B. Welch, "Thinking about Availability in Large Service Infrastructures," *16th Workshop on Hot Topics in Operating Systems*, 2017, pp. 12–17.
8. R. Harper et al., "DynaPlan: Resource placement for application level clustering," *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops*, 2011, pp. 271–277.
9. D.J. Scales, M. Nelson, and G. Venkitachalam, *The design and evaluation of a practical system for fault-tolerant virtual machines*, technical report VMWare-RT-2010-001, VMWare, Inc., May 2010.
10. V. Salapura, R. Harper, and M. Viswanathan, "ResilientVM: high performance virtual machine recovery in the cloud," *Proceedings of the 1st International Workshop on Automated Incident Management in Cloud*, 2015, pp. 7–12.
11. V. Salapura and R. Harper, "Remote Restart for a High Performance Virtual Machine Recovery in a Cloud," *IEEE 8th International Conference on Cloud Computing* (CLOUD 15), 2015.

ABOUT THE AUTHORS

Valentina Salapura is an IBM Master Inventor and a system architect at IBM Watson Health. Her research interests are high-performance computing, resilient cloud computing, and parallel computing. Salapura received a PhD in computer science from Technische Universität Wien. She is an ACM Distinguished Speaker and an IEEE Fellow. Contact her at salapura@us.ibm.com.

Rick Harper is a research staff member at IBM Watson Health. He focuses on fault tolerance, high availability, disaster recovery, distributed computing, problem determination, problem prediction, systems management, workload optimization, and cloud computing. Harper received a PhD in computer and aerospace engineering from MIT. Contact him at reharper@us.ibm.com.

CUP: A Formalism for Expressing Cloud Usage Patterns for Experts and Nonexperts

Aleksandar Milenkoski
ERNW

Alexandru Iosup
Vrije Universiteit Amsterdam and Delft University of Technology

Samuel Kounev
University of Würzburg

Kai Sachs
Careem

Diane E. Mularz
MITRE

Jonathan A. Curtiss
MITRE

Jason J. Ding
Salesforce.com

Florian Rosenberg
IBM Thomas J. Watson Research Center

Piotr Rygielski
SAP SE

The proliferation of cloud services, from infrastructure servers to software, has led to new patterns in service deployment and provisioning practices, but not to standards for expressing and communicating such patterns to a broad-based audience. To enable the formal description and pattern classification of scenarios where cloud services are combined and provisioned to end users, we propose the Cloud Usage Patterns (CUP) formalism. With CUP, both general end users and cloud experts can express patterns, textually and visually. By expressing patterns seen in practice, we demonstrate that CUP is practically useful and makes the use of lengthy prose descriptions obsolete, which is currently common and often results in misunderstandings.

Cloud computing enables the on-demand provisioning of services, leading to time and cost efficiency. Spurred by technology advances and by the emergence of major cloud service providers, such as Amazon, Microsoft, and Alibaba, a variety of cloud services and processes for service deployment and provisioning to end users have emerged. Cloud service deployment and provisioning involve different types of service delivery models, providers, and stakeholders. Such services, and the processes where cloud services are combined in service

Copublished by the IEEE CS and IEEE ComSoc
2325-6095/18/$33.00 USD ©2018 IEEE

chains to provide value to end users, are ripe to be described in patterns,[1–2] which we refer to as *cloud usage patterns*.

Cloud services are provisioned under service delivery models that bundle and abstract a set of service functionalities, or *abstraction levels*. NIST differentiates between three abstraction levels: *infrastructure as a service* (IaaS), *platform as a service* (PaaS), and *software as a service* (SaaS).[3] IaaS enables the provisioning of basic infrastructure resources (e.g., servers, networks, and storage) such that an end user may deploy arbitrary software, starting with the OS. PaaS enables end users to deploy and host applications developed using libraries, services, languages, or tools provided by a cloud provider; a PaaS end user typically does not access directly the underlying infrastructure. SaaS enables end users to use applications but excludes control over infrastructure and application-hosting environments.

We distinguish between *native* and *nonnative* cloud service providers. A native cloud provider is an administered entity (i.e., an entity whose boundaries are defined with the boundaries or jurisdiction of given operation management policies) that owns cloud infrastructure (e.g., a datacenter). A nonnative cloud provider is an administered entity that does not own cloud infrastructure, but relies on provisioned resources from native cloud service providers in order to provide services to consumers. Thus, a nonnative cloud provider has the role of both a service provider and consumer. A provider offers services to consumers and may be a native or a nonnative cloud provider that operates at any of the abstraction levels. A consumer may be either a cloud provider operating at one of the abstraction levels and consuming resources at the same time, or an end user.

We also distinguish between two types of stakeholders: *end users* and *organizations*. An end user (an individual or a system) consumes cloud services provided by a native or nonnative cloud provider. An organization is an entity that may be either the organization to which an end user belongs or a cloud service provider.

Enabling debates about cloud service deployment and provisioning practices using intuitive descriptions and classifications is crucial, as such debates facilitate, among other things, sharing best practices and designing new cloud services. We propose in this article *CUP* (Cloud Usage Patterns), a textual and visual formalism for expressing cloud usage patterns. CUP natively offers textual- and visual-language constructs enabling the expression of service chains involving multiple stakeholders and of relationships between the stakeholders (e.g., consumer and provider). The stakeholders consume or provide resources (at orders of magnitude) and operate at a given abstraction level (i.e., IaaS, PaaS, or SaaS). CUP also offers a mechanism allowing the expression of arbitrary, user-defined aspects of service deployment and provisioning (e.g., authorization policies, network properties, and so on).

In contrast to previous work, CUP is designed to service a general audience at the intersection of the research, industry, and user communities, by providing a different tradeoff between pronounceability, expressiveness, and complexity.

CUP is currently the only formalism designed to be pronounceable, which allows for fast verbal discussions about cloud service deployment practices and approaches.

CUP is less verbose than the existing specification languages for describing cloud service deployment and provisioning practices, such as OASIS TOSCA (Topology and Orchestration Specification for Cloud Applications).[4] A highly technical description, such as OASIS TOSCA, excludes common users and inevitably lengthens discussions. CUP has been specifically designed for the purpose of describing a given cloud service deployment and provisioning practice or scenario. In summary, CUP is to full-blown specification languages what domain-specific languages (DSLs—e.g., Hadoop Pig) are to full-blown programming languages (e.g., Hadoop code written in Java). We believe that by using CUP, potential and actual cloud users can easily collaborate in the specification of service-provisioning requirements, cloud system designers can identify and share best practices, and researchers and consultants can engage in pattern classification and comparison.

The formalism presented in this article has been developed by the Cloud Working Group[5] of the Standard Performance Evaluation Corporation (SPEC) and officially endorsed by the Research

Group of SPEC. In this article, we begin with an example where CUP is not used, to show the practical need for CUP. We then introduce CUP; we compare CUP with popular specification languages, such as OASIS TOSCA; and we show how CUP can be used in practice. Finally, we discuss the future of CUP.

WITHOUT THE CUP FORMALISM

Consider the following cloud-service-provisioning scenario. StartupSaaS is a provider of popular social-game applications located in the US. Since its inception, StartupSaaS has used infrastructure resources leased from ExcellentIaaS in order to deploy and run gaming services. However, owing to the need for finer granularity in resource management and customization, StartupSaaS eventually built its own IaaS cloud. In addition, StartupSaaS still leases additional infrastructure resources from ExcellentIaaS when the capacity of its IaaS cloud is saturated.

Infrastructure resources are provisioned to StartupSaaS in the unit of a virtual machine (VM) of size L (large), which is how ExcellentIaaS names VMs with a 4-GHz CPU, 4 Gbytes of main memory (RAM), and 60-Gbyte hard disk space (HDD). ExcellentIaaS delivers resources to StartupSaaS only when StartupSaaS reports CPU and memory utilization higher than 80 percent of all infrastructure resources that it owns. The amount of resources leased to StartupSaaS increases exponentially while StartupSaaS's resources are saturated and decreases linearly if StartupSaaS reports that the capacity of its infrastructure is not saturated anymore. ExcellentIaaS owns datacenters in both Europe and the US, and it provides infrastructure resources to StartupSaaS from its datacenter in the US for the sake of efficiency.

How to describe this scenario with a simple depiction and/or a string of just several characters? Would such a description ease discussions between SaaS and IaaS experts from StartupSaaS and ExcellentIaaS when combining expertise to jointly provide better services to gamers?

Sometimes StartupSaaS cannot offer a good-quality experience to its players. In such cases, explaining to the players what went wrong is part of the commercial strategy of StartupSaaS, which must be done convincingly yet without excessive technical detail. Furthermore, to propose improvements and to assess StartupSaaS's service-provisioning practice, researchers and independent assessors need to exchange relevant information with the engineers at StartupSaaS without using lengthy prose. *Would the simple formalism be helpful here?*

As mentioned earlier, StartupSaaS has changed its service-provisioning practice from exclusively renting resources from ExcellentIaaS to building an in-house cloud while continuing to rent resources from ExcellentIaaS. *How can the simple formalism enable the different business units within StartupSaaS to efficiently identify a promising practice, compare it with best practices in its own and other industries, and identify the changes needed to improve the service provided to their customers?*

Answering these questions is the essence of our CUP formalism.

WITH THE CUP FORMALISM

CUP enables the specification of each participant in a given service chain, from the owner of virtualized or nonvirtualized hardware, to the value-adding cloud service providers operating at the different abstraction levels, to end users. CUP supports the expression of *elementary patterns*, where only a single provider at any abstraction level provisions resources to a consumer, and *extended patterns* involving, for example, hybrid resource provisioning or mediators, which we discuss later. CUP supports the expression of cloud usage patterns in both a textual and visual form.

We designed CUP to satisfy the following requirements:

- *Expressiveness*. The formalism should be expressive enough to enable the expression of any pattern seen in practice.
- *Comprehensibility*. A pattern expression should be intuitive and easy to understand.

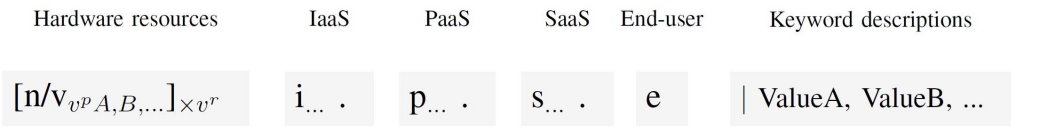
- *Nonambiguity*. The process for expressing a cloud usage pattern should be clearly defined and produce distinct results for different cloud usage patterns (i.e., a single pattern expression should express only a single, distinct cloud usage pattern).

We summarize in the following the main features of CUP; for more details, we refer the reader to *Cloud Usage Patterns: A Formalism for Description of Cloud Usage Scenarios*.[6]

The Textual CUP Formalism

The expression of a cloud usage pattern with the textual CUP formalism is a string, which we refer to as a *CUP string*. A CUP string consists of the sections *Hardware resources*, *IaaS*, *PaaS*, *SaaS*, *End user*, and *Keyword descriptions*, some of which correspond to the different abstraction levels (see this article's introduction).

Figure 1 depicts the expanded format of a CUP string for expressing elementary patterns (read from left to right; optional characters are depicted). Descriptions of the textual-language constructs depicted in Figure 1 are presented in Table 1. In Figure 2a, we present several example CUP strings that we refer to in this section and in the section "The Visual CUP Formalism."



where,

$v^{p/r}$ is "$value_1 - value_2$" | $value_{1/2} \in N^+ \wedge value_1 < value_2$;

A/B is "$cst : KeyA/B/...$", where
$KeyA/B/...$ is a unique string in $\{S^* \setminus \{\varepsilon\}\} \mid S = \{'a', ..., 'z', 'A', ..., 'Z', '0', ..., '9'\}$;

ValueA/B is "$KeyA/B/... = \{value_{A/B/...}\}$", where
$value_{A/B/...}$ is a unique string in $\{A_p^* \setminus \{\varepsilon\}\}$.

Figure 1. Expanded format of a CUP (Cloud Usage Patterns) string for expressing elementary patterns. (ε is an empty string; $*$ is the Kleene operator; $N+$ is the set of positive natural numbers; A_p is the set of ASCII printable characters; and the sections "IaaS," "PaaS," and "SaaS" have the same format as "Hardware resources.")

Table 1. An overview of the textual-language constructs for expressing cloud usage patterns in textual form.

Textual-language construct	Description
n	Denotes hardware provisioning without the use of virtualization technology
v	Denotes hardware provisioning with the use of virtualization technology
i	Denotes a cloud provider at the IaaS abstraction level (optional)
p	Denotes a cloud provider at the PaaS abstraction level (optional)
s	Denotes a cloud provider at the SaaS abstraction level (optional)
e	Denotes an end user
.	Denotes the crossing of a boundary between two separate stakeholders and administered entities in a service-provisioning relationship—i.e., an external service-level agreement (optional)
v^p	Used for specifying an amount of resources provisioned by a provider—i.e., provisioning volume (optional; v^p is a single value or a range of values)
[ ]	Used for encapsulating an expression specifying replicated resource-provisioning capabilities—i.e., a replica (optional)
$\times v^r$	Used for specifying a number of replicas (optional; v^r is a single value or a range of values)
-	Used for specifying v^p and/or v^r as value ranges (optional)
cst:	Custom; used for defining keywords (e.g., unit acronyms) not deemed standard by communicating parties (optional)
KeyA/B/ ...	User-defined keywords—keys (optional)
|	Used to mark the beginning of the "Keyword descriptions" section (optional)
{ }	Used for encapsulating the value of a key (optional)
()	Used for encapsulating pattern expressions (optional)

a)

CUP string name	CUP string
CUP A	vips.e
CUP B	$vi_{5-1000\,VM}$.s.e
CUP C	$vi_{5-1000\,cst:\,L}$.s.e | L={4GHz CPU, 4GB RAM, 60GB HDD}
CUP D	(vi)(vi.)s.e
CUP E	(vi.)i.e
CUP F	(vip.)(vi.p.)p.s.e
CUP G	vi.e
CUP H	ni.s.e
CUP I	$(ni_{1-100\,cst:VM}.)(vi_{1-100\,cst:VM}.)$s.e | VM={2GHz CPU, 3GB RAM, 60GB HDD}
CUP J	$(ni_{1-100\,cst:VM}.)(vi_{1-100\,cst:VM}.)$i.s.e | VM={2GHz CPU, 3GB RAM, 60GB HDD}

b)

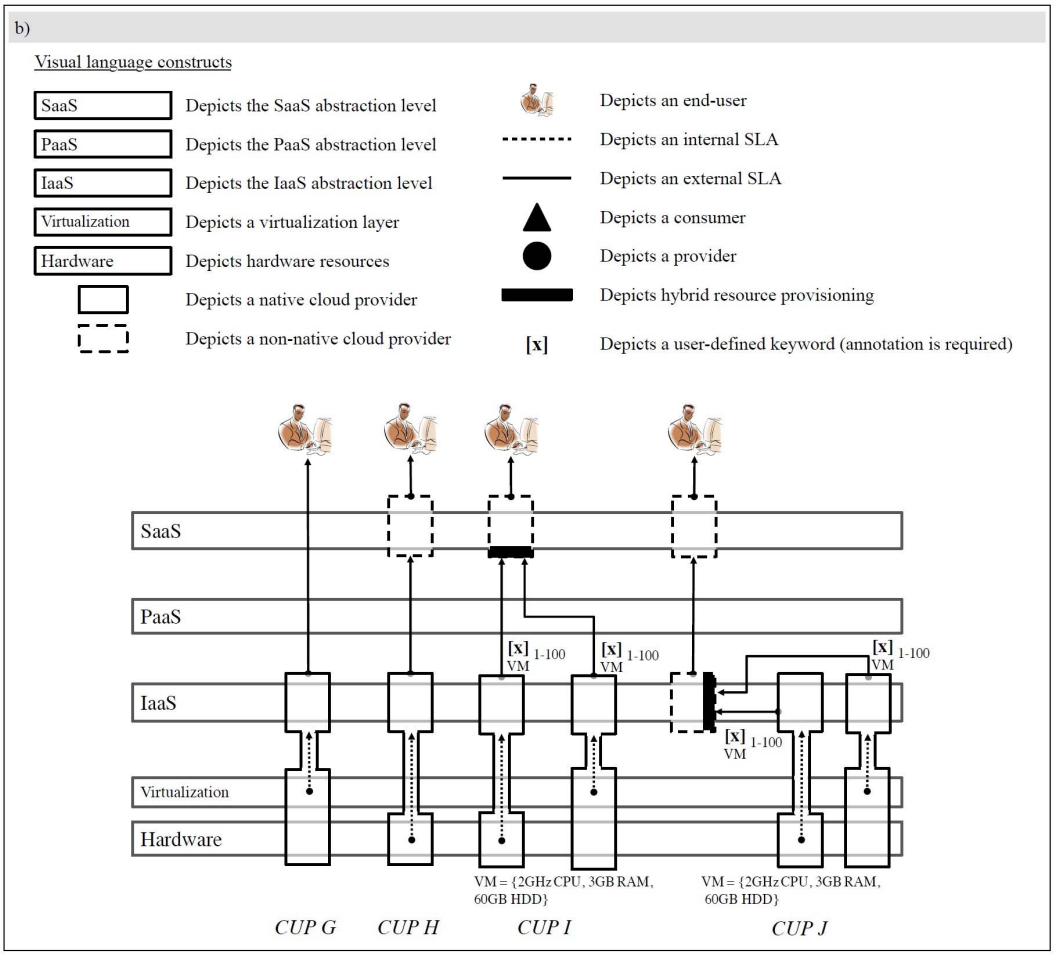


Figure 2. (a) Example CUP strings. (b) Visual-language constructs for visualizing CUP strings, and visual forms of the CUP strings *CUP G*, *CUP H*, *CUP I*, and *CUP J*.

A CUP string consists of at least three sections, including the sections "Hardware resources" and "End user," and at least one of the sections "IaaS," "PaaS," and "SaaS." The order of the letters denotes the provider–consumer pairs that exist in a given pattern. For instance, the CUP string *CUP A* (see Figure 2a) denotes the following provider–consumer pairs: (*Hardware resources, IaaS*), (*IaaS, PaaS*), (*PaaS, SaaS*), and (*SaaS, End user*).

A *dot* (".") marks the crossing of a boundary between a provider and consumer that are two separate stakeholders and administered entities (i.e., cloud service providers, or a cloud service provider and an end user; see the introduction). In practice, this is regulated by a service-level agreement (SLA) in which quality-of-service (QoS) requirements and customer contracts are defined. In case a boundary between two separate administered entities is crossed, we refer to the SLA between the provider and consumer as an *external SLA*. In case such a boundary is not crossed, we refer to the SLA between the provider and consumer as an *internal SLA*.

CUP supports the specification of volumes of resources provisioned by providers (single values or ranges, depicted as v^p in Figure 1). By enabling the specification of v^p as a volume range, CUP supports the specification of elastic properties of cloud providers. That is, we express elasticity, a feature for provisioning an arbitrary amount of resources when needed, by specifying the lowest and the highest amount of resources that may be provisioned by a provider at a given point in time. For instance, the CUP string *CUP B* (see Figure 2a) denotes that the infrastructure provider may provide between 5 and 1,000 units of infrastructure resources to the SaaS provider when needed.

CUP supports the specification of *replicas* (i.e., replicated resource-provisioning capabilities) and their number with the constructs [] and $\times v^r$ (see Table 1). The latter is important since replicas are commonly used in practice—for example, for ensuring service reliability through redundancy or providing for an order-of-magnitude better service performance to users.

We assume that a given amount of resources are provisioned from a provider to a consumer in a unit. A unit is normally specified with an acronym, which may or may not be deemed standard by the different communicating parties. For instance, one normally easily relates "VM" to "virtual machine" as a unit of resources provisioned by an IaaS provider. However, some acronyms might not be known to everyone discussing a service-deployment-and-provisioning scenario. For instance, in the example provided in the section "Without the CUP Formalism," service designers and developers at StartupSaaS might not know what exactly ExcellentIaaS understands regarding "a VM of size L."

To address the above issue, we introduced a key–value mechanism that enables the specification of keywords and associated descriptions as a writer of a CUP string sees fit. This includes unit acronyms and descriptions as well as keywords for specifying various cloud provider properties, such as geographical location and elasticity implementation details. The latter is useful since currently there are many different understandings of what is elasticity.[7]

As opposed to using keywords known by the communicating parties (see, for example, *VM* in *CUP B* in Figure 2a), we require that any keywords that need to be described are preceded by the keyword *cst:*. *CUP C* (see Figure 2a) demonstrates the definition and description of a unit "virtual machine of size L" (see the section "Without the CUP Formalism").

CUP supports the expression of extended cloud usage patterns involving hybrid resource provisioning or mediators. Hybrid resource provisioning is provisioning of resources to a consumer from multiple native or nonnative cloud providers at the same abstraction level (see, for example, the section "Without the CUP Formalism"). For example, in *CUP D* (see Figure 2a), the pattern expressions (i.e., parts of a complete CUP string) *vi* and *vi.* encapsulated in parentheses express the patterns of the two different IaaS providers that provide infrastructure resources to the SaaS provider.

CUP supports the expression of value chains with mediators. In the context of cloud computing, a value chain is a network of service providers that cooperate in order to add or generate value for end users.[8] An example of a mediator is an organization that leases platform resources from one or more PaaS providers, adds value to the leased services (e.g., by adding functionalities), and offers them to consumers as its own product.

When expressing value chains with mediators, pattern expressions describing service provisioning to mediators are encapsulated in parentheses (see *CUP E* and *CUP F* in Figure 2a). For instance, *CUP E* expresses a pattern where a mediator provides infrastructure resources to end users. The infrastructure resources are leased from an IaaS provider. *CUP F* expresses a pattern involving both hybrid resource provisioning and a mediator.

The Visual CUP Formalism

The visual CUP formalism is a recommendation for visualizing CUP strings in an intuitive manner and in a form easy to understand by a wide audience. CUP strings can be visualized using the visual-language constructs presented in Figure 2b. In Figure 2b, we also depict four examples of visualized CUP strings.

The CUP string *CUP G* (see Figure 2a) expresses a pattern where a native IaaS cloud provider delivers infrastructure resources to an end user. The native cloud provider, depicted as a box with a solid line in Figure 2b, and the end user are separate administered entities. Thus, the QoS requirements and customer contracts between the provider and the end user are defined as part of an external SLA visualized with a solid line. The underlying hardware resources and the IaaS abstraction level belong to the same administered entity (i.e., the IaaS provider). Therefore, the QoS requirements and customer contracts for the provisioning of hardware resources are defined as part of an internal SLA visualized with a dashed line. The hardware resources are provisioned with the use of virtualization technology. This is reflected by placing the graphical symbol of a provider inside the box depicting the virtualization layer.

The visual CUP formalism matches its textual counterpart to only a certain extent. Definitions and descriptions of user-defined keywords (done with a key–value mechanism when writing CUP strings, see the section "The Textual CUP Formalism") and specifications of values or value ranges (v^p and v^r; see Figure 1) obviously cannot be visualized in a standardized manner using visual-language constructs for that purpose. Therefore, we recommend the annotation of CUP string depictions with keywords, keyword descriptions, values, and value ranges.

In Figure 2b, the depictions of the CUP strings *CUP I* and *CUP J* (see Figure 2a) demonstrate the use of the graphical element [x] for depicting a user-defined keyword (i.e., VM; see Table 1). They also demonstrate annotating CUP string depictions with keywords, keyword descriptions, and value ranges, which is required when [x] is used. Finally, the depictions of *CUP I* and *CUP J* demonstrate visualizing CUP strings that express patterns involving hybrid resource provisioning and mediators, respectively. For clear depiction, the graphical element visualizing hybrid resource provisioning can be placed horizontally (*CUP I*) or vertically (*CUP J*).

AN EMPIRICAL COMPARISON OF CUP WITH OTHER SPECIFICATION LANGUAGES

We compare here CUP with OASIS TOSCA in the XML and YAML (Yet Another Multicolumn Layout) formats (see the introduction). We also compare CUP with the commercial specification language Amazon Web Services CF (CloudFormation) in the JSON (JavaScript Object Notation) format.

We compare the sizes of text snippets used to express cloud usage patterns. Cloud usage pattern expressions of smaller sizes are preferable, not only for preserving space and messaging bandwidth, but also for quicker parsing by human eyes. In addition, pattern expressions with sizes in the order of a few tens of characters, or fewer, can be pronounced quickly. In order to compose expressions of minimal sizes when expressing patterns with OASIS TOSCA and Amazon CF, we applied the best practices listed in the TOSCA Primer and the documentation of Amazon CF.

The results of our study are summarized in Table 2. We consider three scenarios: a single server hosted by an IaaS cloud ("1 server"), 10 servers hosted by an IaaS cloud ("10 servers"), and two servers provisioned by two separate IaaS providers ("2 servers, hybrid provisioning"). We consider two cases for each scenario, where a server configuration is specified ("configuration") and not specified ("no configuration"). In Table 2, the column "CUP string" presents the CUP strings for the considered scenarios, whereas the column "CUP" presents the sizes of these strings. We do not present here the cloud usage pattern expressions in the OASIS TOSCA and Amazon CF languages owing to their sizes, some of which are up to thousands of characters (see Table 2). These expressions are available online for reference at http://tinyurl.com/orw72jq.

Table 2. A comparison of CUP with other specification languages.

Considered scenarios and CUP strings	
Scenario	CUP string
1 server (no configuration)	vi.e
1 server (configuration)	$vi_{cst:L}$.e | L={3GHz GPU, 4GB RAM, 60GB HDD}
10 servers (no configuration)	vi_{10}.e
10 servers (configuration)	$vi_{10cst:L}$.e | L={3GHz GPU, 4GB RAM, 60GB HDD}
2 servers, hybrid provisioning (no configuration)	(vi.)(vi.)e
2 servers, hybrid provisioning (configuration)	($vi_{1cst:L}$.)($vi_{1cst:L}$.)e | L={3GHz GPU, 4GB RAM, 60GB HDD}

Sizes (in number of characters) of expressions of cloud usage patterns				
	Cloud Usage Pattern expressions			
Scenario	CUP	Amazon CF (JSON)	OASIS TOSCA (YAML)	OASIS TOSCA (XML)
1 server (no configuration)	4	1,009	111	469
1 server (configuration)	36	522	188	554
10 servers (no configuration)	6	1,569	543	N/A
10 servers (configuration)	38	1,082	1,342	N/A
2 servers, hybrid provisioning (no configuration)	11	2,286	350	N/A
2 servers, hybrid provisioning (configuration)	50	1,562	503	N/A

We find that CUP is the most compact representation and the only one that is pronounceable when simple patterns are expressed (see, for example, the scenario "1 server (no configuration)" in Table 2). In comparison to CUP, Amazon CF and OASIS TOSCA are much more verbose, which becomes evident by looking at the number of characters needed to express the considered cloud usage patterns (see Table 2). Furthermore, the OASIS TOSCA XML-based format is less expressive than CUP for complex cloud usage patterns, such as patterns describing hybrid resource provisioning, for which there is no standard approach in the format. In addition, we observed that the OASIS TOSCA YAML-based format is repetitive. For example, adding more servers (even identical; see for example, the "10 servers" scenarios in Table 2) increases significantly the size of the pattern expressions in this format.

To demonstrate the compactness, pronounceability, and practical usefulness of CUP, we surveyed real-world cloud usage patterns and expressed them using CUP. For descriptions of the

applied method for selecting these patterns and more information on them, we refer the reader to "Cloud Usage Patterns: A Formalism for Description of Cloud Usage Scenarios."[6]

We depict in Figure 3 the CUP strings that express the considered patterns, and their visual forms. Each pattern expressed with CUP bares a code name based on the full name of the provider that provides resources to end users—i.e., AWS (Amazon Web Services), FBK (Facebook), GAN (GoAnimate), EJT (easyJet), EZS (EZasset), FRC (Force.com), SFR (Salesforce.com), DNB (DenizBank), ZNG (Zynga), and DTO (Dito). For instance, Force.com is a PaaS provider and provides platform resources to end users conforming to the pattern *vp.e* (the FRC scenario). Conforming to the pattern *vps.e* (the SFR scenario), Salesforce.com offers a customer-relationship-management application to end users, whose development and hosting is supported by platform resources provisioned by Force.com.

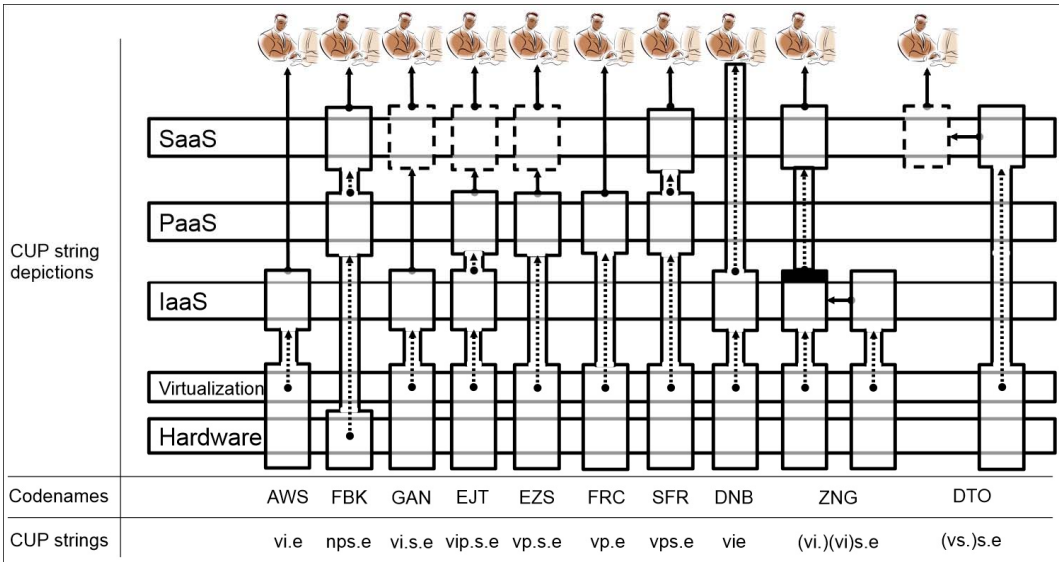


Figure 3. CUP strings expressing real-world cloud usage patterns and their visual forms. For an explanation of the code names, see the main text.

We find that CUP fills a missing space among languages offered by current cloud providers, such as Amazon CF. Commercial cloud providers working with broad consumer bases (e.g., Amazon, Google, and Microsoft) have simplified their formulation of services and SLAs by breaking them down into individual components. For example, Amazon Web Services offers individual, unit-sized services, such as leasing a single VM or an integral multiple thereof. However, this approach leaves all the difficulty of composing a complete service to the user and does not facilitate dialogue between service providers and consumers.

On the other hand, when cloud providers work with a narrow set of customers, they use detailed contracts agreed upon in lengthy meetings. For example, financial institutions are now increasingly using cloud services with specific terms of contract negotiated extensively.

CUP aims to simplify the discussion and agreement in both cases by proposing a formal language allowing for standardized, brief, and often pronounceable descriptions of cloud service deployment and provisioning practices.

CONCLUSION

Addressing the need for a simple and compact, yet expressive formalism for expressing patterns in cloud service deployment and provisioning practices (i.e., cloud usage patterns), we introduced in this article the CUP formalism. CUP is designed for communicating cloud usage patterns in an intuitive manner and in a form easy to understand by a broad-based audience.

We foresee many practical applications of CUP, some of which are the following:

- *Basic agreement between a cloud provider and end users*. A brief specification (i.e., a pattern expression) can be used to define a pre-agreement of service.
- *Researchers describing their problem or experimental setup*. CUPs could be gainfully employed in studies about cloud service deployment and provisioning approaches, especially if there is a sufficient population of examples with correlated data (e.g., cost or performance). For instance, by correlating cost data with a variety of CUPs, the community could learn about and quantify savings of one pattern over another.
- *Acquisition process and selection of cloud offerings*. For service deployment managers and government procurement procedures, patterns expressed with CUP can be used for the automated screening of cloud service offerings. For instance, managers could use a cloud broker to select among many hundreds of offerings, especially in the well-defined cloud-infrastructure-service domain.

There are a number of ways in which this work could be continued. For instance, the tradeoff between the expressive power and length of CUP expressions could be investigated through case studies, which may lead to extending and/or adapting the existing formalism. Further, the use of CUP for identifying antipatterns in industrial use cases could be investigated. Finally, with the involvement of the community, a taxonomy for expressing patterns containing, for example, definitions of machine sizes, SLAs, and elasticity policies could be assembled. This effectively would be an effort to build a cloud-usage-pattern knowledge base.

REFERENCES

1. E.C. Withana and B. Plale, "Usage Patterns to Provision for Scientific Experimentation in Clouds," *IEEE Second International Conference on Cloud Computing Technology and Science* (CloudCom 10), 2010, pp. 226–233.
2. *Defining a framework for cloud adoption*, white paper, IBM Global Technology Services, 2010.
3. P. Mell and T. Grance, *The NIST Definition of Cloud Computing*, technical report, July 2009.
4. *OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA)*; https://www.oasis-open.org/committees/tc home.php?wg abbrev=tosca.
5. *RG Cloud Working Group: SPEC Research Group*; http://research.spec.org/working-groups/rg-cloud-workinggroup.html.
6. A. Milenkoski et al., *Cloud Usage Patterns: A Formalism for Description of Cloud Usage Scenarios*, technical report SPEC-RG-2013-001 v. 1.0, April 2013.
7. N.R. Herbst, S. Kounev, and R. Reussner, "Elasticity in Cloud Computing: What It Is, and What It Is Not," *Proceedings of the 10th International Conference on Automatic Computing* (ICAC 13), 2013, pp. 23–27.
8. S. Leimeister et al., "The Business Perspective of Cloud Computing: Actors, Roles and Value Networks," *Proceedings of the 18th European Conference on Information Systems* (ECIS 10), 2010.

ABOUT THE AUTHORS

Aleksandar Milenkoski is an IT security analyst at ERNW. His research interests are in the evaluation of cloud-based systems, with a focus on intrusion detection systems. Milenkoski received his doctorate from the University of Würzburg. Contact him at amilenkoski@ernw.de.

Alexandru Iosup is a tenured full professor and University Research Chair at Vrije Universiteit Amsterdam, and an associate professor at the Delft University of Technology. His research focuses on massivizing (distributed) computing systems—e.g., cloud computing and

big data systems. Applications include big science, business-critical workloads, online gaming, and large-scale education. Iosup received a PhD in computer science from the Delft University of Technology. He's a member of IEEE and ACM. Contact him at a.iosup@vu.nl.

Samuel Kounev is a full professor and the chair of software engineering at the University of Würzburg. His research focuses on the engineering of dependable and efficient software systems, including software design, modeling, and architecture-based analysis; systems benchmarking and experimental analysis; and autonomic and self-aware computing. Kounev received a PhD in computer science from TU Darmstadt. He's a member of IEEE and ACM. Contact him at samuel.kounev@uni-wuerzburg.de.

Kai Sachs is an engineering manager at Careem. His research interests include software engineering, cloud computing, and performance modeling and evaluation. Sachs received a PhD in computer science from TU Darmstadt. Contact him at kai.sachs@careem.com.

Diane E. Mularz is a principal software systems engineer at the MITRE Corporation. Her research interests are in performance and software engineering, patterns, and complex systems. Contact her at mularz@mitre.org.

Jonathan A. Curtiss is a performance and modeling engineer at the MITRE Corporation. He's active in cloud performance modeling and system performance engineering. Curtiss received his BSc in electrical engineering from the State University of New York. He's a Senior Member of IEEE. Contact him at jcurtiss@mitre.org.

Jason J. Ding is a senior director of the Performance Engineering Department at Salesforce.com, where he leads performance-engineering teams working on the performance and scalability of cloud-based enterprise applications and search solutions. Ding received a PhD in computer science from Texas A&M University. Contact him at jding@salesforce.com.

Florian Rosenberg is a manager and research staff member at the IBM Thomas J. Watson Research Center. His research interests are in DevOps and cloud computing, particularly software configuration and continuous deployment. Rosenberg received a PhD from the Vienna University of Technology. He's a member of IEEE and ACM. Contact him at rosenberg@us.ibm.com.

Piotr Rygielski is a developer at the SAP Innovation Center. His research interests are in modeling and performance analysis of virtualized network infrastructures in cloud datacenters. Rygielski received a PhD from the University of Würzburg. He's a member of IEEE. Contact him at p.rygielski@sap.com.

IEEE computer society

PURPOSE: The IEEE Computer Society is the world's largest association of computing professionals and is the leading provider of technical information in the field.
MEMBERSHIP: Members receive the monthly magazine *Computer*, discounts, and opportunities to serve (all activities are led by volunteer members). Membership is open to all IEEE members, affiliate society members, and others interested in the computer field.
COMPUTER SOCIETY WEBSITE: www.computer.org
OMBUDSMAN: Direct unresolved complaints to ombudsman@computer.org.
CHAPTERS: Regular and student chapters worldwide provide the opportunity to interact with colleagues, hear technical experts, and serve the local professional community.
AVAILABLE INFORMATION: To check membership status, report an address change, or obtain more information on any of the following, email Customer Service at help@computer.org or call +1 714 821 8380 (international) or our toll-free number, +1 800 272 6657 (US):

- Membership applications
- Publications catalog
- Draft standards and order forms
- Technical committee list
- Technical committee application
- Chapter start-up procedures
- Student scholarship information
- Volunteer leaders/staff directory
- IEEE senior member grade application (requires 10 years practice and significant performance in five of those 10)

PUBLICATIONS AND ACTIVITIES
***Computer*:** The flagship publication of the IEEE Computer Society, *Computer*, publishes peer-reviewed technical content that covers all aspects of computer science, computer engineering, technology, and applications.
Periodicals: The society publishes 13 magazines, 19 transactions, and one letters. Refer to membership application or request information as noted above.
Conference Proceedings & Books: Conference Publishing Services publishes more than 275 titles every year.
Standards Working Groups: More than 150 groups produce IEEE standards used throughout the world.
Technical Committees: TCs provide professional interaction in more than 30 technical areas and directly influence computer engineering conferences and publications.
Conferences/Education: The society holds about 200 conferences each year and sponsors many educational activities, including computing science accreditation.
Certifications: The society offers two software developer credentials. For more information, visit www.computer.org/certification.

NEXT BOARD MEETING

7-8 June 2018, Phoenix, AZ, USA

EXECUTIVE COMMITTEE

President: Hironori Kasahara
President-Elect: Cecilia Metra; **Past President:** Jean-Luc Gaudiot; **First VP, Publication:** Gregory T. Byrd; **Second VP, Secretary:** Dennis J. Frailey; **VP, Member & Geographic Activities: Forrest Shull**; **VP, Professional & Educational Activities: Andy Chen**; **VP, Standards Activities:** Jon Rosdahl; **VP, Technical & Conference Activities:** Hausi Muller; **2018-2019 IEEE Division V Director: John Walz**; **2017-2018 IEEE Division VIII Director:** Dejan Milojicic; **2018 IEEE Division VIII Director-Elect:** Elizabeth L. Burd

BOARD OF GOVERNORS

Term Expiring 2018: Ann DeMarle, Sven Dietrich, Fred Douglis, Vladimir Getov, Bruce M. McMillin, Kunio Uchiyama, Stefano Zanero
Term Expiring 2019: Saurabh Bagchi, Leila DeFloriani, David S. Ebert, Jill I. Gostin, William Gropp, Sumi Helal, Avi Mendelson
Term Expiring 2020: Andy Chen, John D. Johnson, Sy-Yen Kuo, David Lomet, Dimitrios Serpanos, Forrest Shull, Hayato Yamana

EXECUTIVE STAFF

Executive Director: Angela R. Burgess
Director, Governance & Associate Executive Director: Anne Marie Kelly
Director, Finance & Accounting: Sunny Hwang
Director, Information Technology & Services: Sumit Kacker
Director, Membership Development: Eric Berkowitz
Director, Products & Services: Evan M. Butterfield

COMPUTER SOCIETY OFFICES

Washington, D.C.: 2001 L St., Ste. 700, Washington, D.C. 20036-4928
Phone: +1 202 371 0101 • **Fax:** +1 202 728 9614
Email: hq.ofc@computer.org
Los Alamitos: 10662 Los Vaqueros Circle, Los Alamitos, CA 90720 **Phone:** +1 714 821 8380
Email: help@computer.org

MEMBERSHIP & PUBLICATION ORDERS
Phone: +1 800 272 6657 • **Fax:** +1 714 821 4641 • Email: help@computer.org
Asia/Pacific: Watanabe Building, 1-4-2 Minami-Aoyama, Minato-ku, Tokyo 107-0062, Japan
Phone: +81 3 3408 3118 • **Fax:** +81 3 3408 3553
Email: tokyo.ofc@computer.org

IEEE BOARD OF DIRECTORS

President & CEO: James Jefferies
President-Elect: Jose M.F. Moura
Past President: Karen Bartleson
Secretary: William P. Walsh
Treasurer: Joseph V. Lillie
Director & President, IEEE-USA: Sandra "Candy" Robinson
Director & President, Standards Association: Forrest D. Wright
Director & VP, Educational Activities: Witold M. Kinsner
Director & VP, Membership and Geographic Activities: Martin Bastiaans
Director & VP, Publication Services and Products: Samir M. El-Ghazaly
Director & VP, Technical Activities: Susan "Kathy" Land
Director & Delegate Division V: John W. Walz
Director & Delegate Division VIII: Dejan Milojičić