The Methods of Cloud Computing

Lauritz Thamsen[1], Jossekin Beilharz[2], Andreas Polze[2], and Odej Kao[1]

[1] Technische Universität Berlin, Germany
[2] Hasso Plattner Institute, University of Potsdam, Germany

Computing, storage, and network resources are available as a utility today through the methods of cloud computing. Users can provision the resources, platforms, and services they need and pay only for their usage. The key technology enabling this is virtualization, yet orchestration and automation tools, distributed application architectures and execution models, as well as higher-level platforms also define how virtual resources are offered and consumed today. At the same time, the rise of cloud computing has been accompanied by cultural shifts, where most notably software development, systems operations, and quality assurance are merging in their roles, processes, and tools. Major trends that have been shaping cloud computing recently include serverless computing, which abstracts resources and resource management entirely from consumers through a fully automatic horizontal application scaling, and the emerging computing paradigms of the Internet of Things, which integrate heterogeneous distributed resources beyond centralized data centers. Meanwhile, current research on cloud infrastructures and operations increasingly employs machine learning to automate resource management and failure handling.

1 Introduction

Cloud computing as a computing paradigm emerged from a history of developments that increased the sharing of available computer resources [94, 112]. Batch systems allowed multiple users to use computers in a time-shared manner, before computers became small and affordable enough to become personal computers. Larger user groups in industry and research, however, quickly outgrew the resources of single computers, leading to the development of clusters. In clusters many computers are connected in a local network, typically hosted in a single room and often equipped with relatively homogeneous resources and software stacks on all nodes. Usually many users within an organization share one or more clusters, so that users reserve a fraction of the entire cluster resources for a particular time. Centralizing the resources into dedicated clusters and sharing access to these clusters makes sense from an economically point of view: centrally housing, operating, and maintaining a larger set of nodes organized in a single cluster can typically be done more efficiently than with decentralized resources. Moreover, statistical multiplexing sharing allows for increased utilization of shared cluster resources, especially when the intensity of workloads varies over time [4].

Computing needs that go beyond the resources of single clusters were addressed by grids, in which heterogeneous and widely distributed resources—be it single computers or entire clusters—are connected to form a unified computing platform. Users of grid systems can submit their jobs, which are then executed by the participating heterogeneous and distributed nodes of the grid. These nodes all run a grid system, managing the resources and effectively abstracting both the heterogeneity as well as the distribution of the nodes.

These developments of resource sharing lead to the vision of computing as a utility, accessible to everyone at self-service, with seemingly unlimited resources available and billed by usage, comparable to how energy and water are made available to customers in the respective grids. The central technology that made this vision of cloud computing a reality is virtualization. It allowed not only to safely run applications of different customers in isolation on the same shared resources in a data center, but to do so with a high degree of efficiency, flexibility in scheduling loads, and tolerance to failures of individual components.

Similar to cluster resource sharing within one organization, yet at orders of magnitude higher scale, cloud providers serve numerous organizations simultaneously, taking advantage of statistical multiplexing and economies of scale: Large numbers of nodes can be acquired, housed, powered, cooled, and maintained much more cost-efficiently by a single large provider than by many separate entities all managing considerably smaller infrastructures. That is, the large size of the data centers of major cloud providers and the large number of customers allow for high degrees of resource utilization, despite varying demands of individual users. In addition, as a value proposition especially for users with varying or growing computing needs, cloud computing offers ways to shift upfront capital expenses to continuous operational costs, having users pay for resource usage instead of the resources themselves.

1.1 The NIST Definition of Cloud Computing

The publication 800-145 of the National Institute of Standards and Technology [80] provides a definition of cloud computing and defines it as "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction". This cloud model is composed of five essential characteristics, three service models, and four deployment models.

The five essential characteristics of cloud computing by this definition are *on-demand self-service, broad network access, resource pooling, rapid elasticity, and measured service*. Users provision resources themselves as needed and without interacting with service provider personell. They access resources and services over networks, using the standard mechanisms and protocols of the Internet. Under the assumption that not all customers require resources simultaneously, schedulers assign the resources currently required from large pools of resources to specific reservations, so any user allocation is provided with physical resources according to its current demand. This is supported by rapid elasticity, which allows to reserve and use exactly as many resources as actually needed over time, reserving additional resources as numbers of requests or queries increase and returning surplus resources as soon as there is less demand. To connect usage to costs, provisioned services are measured and usually billed by reservation time or measured usage. This provides an incentive for providers to offer resources in the first place and for users to only reserve the resources actually needed.

The three service models of the NIST definition are named infrastructure as a service (IaaS), platform as a service (PaaS), and software as a service (SaaS). IaaS describes virtual resource provisioning, where the customer is free to choose—but also responsible for—the operating system, the middleware, the programming environment, and the applications running on top. PaaS balance the responsibilities of customers and providers at a higher level of abstraction than IaaS. The provider promises to take care of scalability, availability, and maintenance of the execution environment, so customers can focus on designing, implementing, and providing their own applications. Thus, customers only control their applications. This advantage comes at a higher operational cost per time-unit and a reduced flexibility regarding the choice of runtime and development environment. SaaS represents a fully transparent environment, where the customer simply uses a particular service without the possibility to run own applications. There are also plenty additional service abbreviations, such as Hardware as a Service or Security as a Service, yet the three initial service models remain defining for cloud computing.

The deployment models described in the NIST definition determine which customer groups are eligible for using a particular cloud infrastructure. A public cloud is available to the general public. It is set up on the premises of a cloud provider, who can be a business, an academic institution, a government organization, or a combination of these. In contrast, a private cloud is dedicated to a single organization, yet the organization is not necessarily owning or operating the cloud. This can also be outsourced to a third party managing the infrastructure on or off premises. Between those two deployment models there are community and hybrid clouds. A community cloud is exclusively used by a specific community of users from organizations that have a shared interest, whereas hybrid clouds are compositions of two or more distinct cloud infrastructures.

1.2 Recents Trends Around Cloud Computing

Virtualization is the enabler of cloud computing, but also leads to new problems. In particular, the ease of provisioning and running virtual servers as needed and without worrying about physical infrastructures often yields server sprawl, as all the virtual machines and replicated application components still need to be managed. The complexity of managing and orchestrating hundreds of virtual servers, applications, components, software versions, dependencies, as well as production and test deployments has been addressed mainly by cultural shifts, specifically the DevOps movement and the idea of infrastructure as code (IaC), yet also by new orchestration tools [10, 85].

Aside of ongoing developments in virtualization technology, with for example OS-level containment features that allow isolated environments with significantly reduced footprints and start-up times, and cultural shifts, which aim at faster and more continuous releases of cloud applications, there are also efforts to further raise the level of abstraction for users with increasingly managed cloud services such as PaaS offerings, including those enabling serverless computing [7]. These cloud services take care of the underlying infrastructure, fault tolerance, scaling and load balancing, while also providing program runtimes in terms of languages, libraries, and external interfaces, allowing users to fully focus on their applications as long as their applications fit with the platform runtimes.

The increasingly connected, sensor-equipped, and distributed devices of the Internet of Things (IoT) pose new challenges for distributed application architectures and execution models. These systems produce sensor data streams, which need to be processed continuously, often with specific quality of service (QoS) requirements such as low end-to-end latencies. This can be achieved with stream processing, following the distributed dataflow model popularized by MapReduce [32]. Such distributed stream processing systems support continuous, low-latency processing of constantly arriving streaming data and provide features to analyze infinite data streams such as windowed aggregations and recovery from partial failures. However, many distributed data management and processing systems currently still expect to be run on centralized and homogenous resources, while new distributed computing paradigms such as edge and fog computing [31, 100] propose to integrate more heterogeneous and dynamic resources such as devices, gateways, and distributed servers outside of centralized data centers. Likely, sharing resources will also be beneficial in these environments, yet efficient and dependable management of dynamic and heterogeneous resources is still a significant challenge for sensor stream processing and other IoT applications.

Outline. The remainder of this report is organized as follows: Section 2 looks at virtualization as the key technology enabling cloud computing. Section 3 discusses tools and practices for managing large sets of virtual resources. Section 4 outlines how fault-tolerant and scalable distributed applications can be run on cloud resources. Section 5 presents central methods and systems for data-intensive applications. Section 6 looks at high-level managed cloud platforms, including serverless computing. Section 7 describes the main ideas of the emerging distributed computing paradigms that connect the IoT to clouds. Section 8 concludes this report with a summary of current research on cloud computing methods, including some of our own results.

2 Foundation of Clouds: Virtual Resources

Before virtualization technology became mature and popular, users and organizations were required to run their business-critical applications and workloads on dedicated physical servers, usually hosted in their own data centers for privacy and security reasons. Virtualization then enabled isolation, high resource utilization, rapid provisioning, and dynamic elasticity. It allowed to safely and efficiently serve the computation and storage needs of external customers with the resources of one's data center. First, virtualization allows to safely host virtual machines of multiple different customers on the same physical resources. Second, it allows a significantly increased amount of flexibility as system virtualization allows not only to run multiple virtual machines on a single physical machine, but each of these virtual machines can be assigned a specific amount of resources and run its own operating system. In comparison to the resources of large cloud providers, dedicated physical resources of single organizations have a much higher chance of being either underutilized or overutilized with the assigned workload. This is true even if a set of applications is optimally consolidated on a fitting set of physical resources. Cloud providers simply have significantly larger optimization spaces in all resource management decisions as they are able to consolidate the workloads of users of multiple distinct organizations, while virtualization furthermore allows to actively balance loads such as by migrating virtual machines from one physical node to another at runtime and with little noticeable impact on application performance.

Virtualization allows to snapshot the state of a virtual server, whereas single dedicated bare-metal servers are often enough single points of failures. With snapshots and live migration servers can be available through planned maintenance and quickly re-instated in case of unforeseen outages. Moreover, with cloud operating systems that manage resources and provide interfaces to users and systems, IaaS providers allow their users to quickly provision virtual resources, including automatic scaling of current reservations.

The central idea of virtualization is to introduce a sandbox in which a customer's application can be executed in isolation and using a specific amount of the available resources. This is true for both system virtualization [108], in which virtual resources come in form of virtual machines, and containerization [82, 109], in which virtual resources are provided as containers.

Figure 1 shows an overview over the three main virtualization approaches.

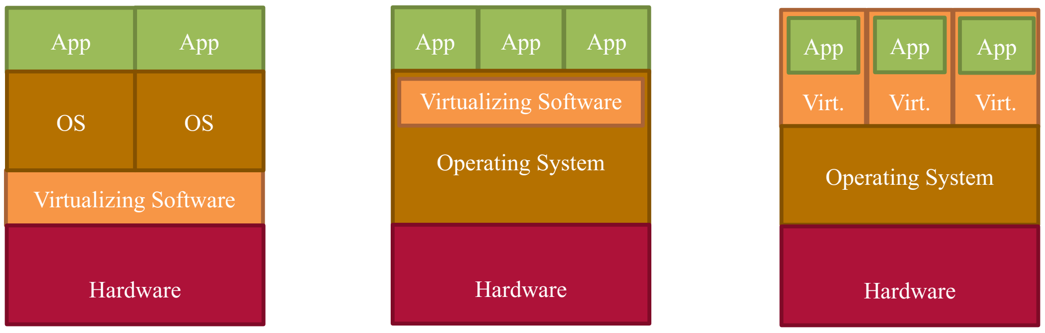

Figure 1: Approaches to virtualization, from left to right: system virtualization, OS-level virtualization, and process-level virtualization.

System virtualization allows multiple guest systems (including the guest operating systems) to run on virtualized hardware. In OS-level virtualization, programs run in virtualized environments called containers. These containers run on the same operating system kernel. Process-level virtualization provides an sandbox and abstraction on top of the operating system APIs for a certain program.

General-purpose virtual resources in cloud environments are provided predominantly through system virtualization, yet increasingly also using OS-level virtualization, while process virtualization is often employed in addition for applications, either by users of virtual resources or in higher-level cloud platforms as part of the provided environment.

2.1 System Virtualization

In system virtualization, access to resources by guest operating systems, which can then run arbitrary user applications, is virtualized. This includes access to main memory and hardware devices. It requires the same instruction set architecture for efficient execution of virtual machines[3], yet allows for different hardware characteristics and different operating systems. System virtualization VMMs are classified as Type-I or Type-II hypervisor. Type-I hypervisors run directly on the hardware, while Type-II hypervisors run on top of a host operating system.

In order to allow any guest operating system to function unchanged within the virtualized environment, it needs to be disempowered without noticing. One simple approach to system virtualization is the idea of trap-and-emulate. If access to resources that need to be virtualized can only happen via privileged instructions, guest programs can be executed natively in unprivileged mode and if a privileged instruction causes a trap, this instruction can be emulated by the VMM. Popek and Goldberg [95] define a set of conditions for a computer architecture to support efficient virtualization. The most important condition is that the sensitive instructions are a subset of the privileged instructions. Sensitive instructions are instructions that either attempt to change some global state of the computer or where the behavior of the instruction is dependent on some global state of the computer. Processor architectures that fulfill these requirements allow the use of the trap-and-emulate approach.

On processor architectures that don't fulfill Popek and Goldbergs requirements for efficient virtualization, like IA-32, a different approach is necessary. One way to virtualize these architectures is binary translation. In binary translation, the program is rewritten by the VMM during execution, so that all sensitive instructions are either replaced by a trapping instruction or directly by an emulation of this instruction. Another approach is paravirtualization, that changes the requirements for the guest system by not allowing any unchanged guest to be executed inside the virtual machine but rather only guest systems that are adapted for this VMM.

[3]While emulating system virtualization allows virtualizing the instruction set architecture, emulation is seldomly used by cloud providers. This is not only due to the increased virtualization overhead, but also due to the availability of most workloads for all architectures.

Prominent examples of system virtualization technologies are VMWare ESXi Server [25], Xen [8], and features for hardware-assisted virtualization such as Intel's Virtualization Technology [120]. For their cloud offerings, Amazon Web Services has been using Xen for a long time, and now uses a KVM-based VMM.

2.2 OS-Level Virtualization

Containers, in contrast to virtual machines, are isolated environments mostly used for single guest applications. A specified process is started and isolated using kernel containment features. Examples for such containment features include namespaces, cgroups, and capabilities in the case of the Linux kernel [61]. Prominent examples for containers on Linux are LXC containers[4] and docker[5] [82]. Besides providing user tools for efficient creation and management for containers, docker also includes a form of declarative description for building new container instances, called dockerfiles, and an ecosystem of public repositories for sharing container images, from which container instances can be started.

In comparison, os-level virtualization allows for smaller images due to the shared operating system kernel, for faster startup of instances as only isolated processes are created for the containerized applications, and also for a higher density on a single host as well as reduced overhead as neither CPU instructions nor resource access need to be virtualized by a hypervisor. Virtual machines on the other hand are currently still the much more mature technology and, at the same time, hypervisors consist of an order of magnitude less code than operating system kernels. From this point of view it is fair to consider virtual machines safer when assessing the current state of affairs [37]. Furthermore, live migration of containers is not yet widely available, whereas live migration of virtual machines is well understood and widely used [37].

2.3 Process-Level Virtualization

Process-level virtualization describes virtual machines used on top of a operating system for a specific class of programs. Programs are translated to an intermediate representation (e.g. bytecode) and are executed inside a VM that abstracts the specifics of the underlying Operating System and hardware. Examples include Java Virtual Machine [70] and the Common Language Runtime [19]. While many PaaS-offerings employ process-level virtualization, it is always combined with system virtualization for security purposes.

[4]`https://linuxcontainers.org/lxc/`, accessed 2022-02-16
[5]`https://docs.docker.com/`, accessed 2022-02-16

3 Management of Virtual Resources: Tools and Practices for Cloud Infrastructures

Virtualization considerably eases the provisioning and running of virtual servers. Users can instantiate and re-start virtual servers as needed and without worrying about the specifics of the physical infrastructure. However, this still leaves ample resource, dependency, and configuration management tasks for users to conduct.

As it becomes easier to provision new, duplicate running, and re-instate previously snapshotted virtual machines and containers, users tend to make use of larger numbers of such appliances and, consequently, also cloud resources. This is referred to as *server sprawl*, pointing to management complexities that directly result from the sheer number of virtual servers and application components started. Moreover, such large sets of virtual resources also tend to collect considerable amounts of technical debt over time through *configuration drift* and *snowflake servers* [85]. That is, IaaS cloud users create virtual machine instances and replicate server application components with a single click or automatically through APIs, yet as they update specific configurations manually over time and without full documentation, it becomes increasingly hard to understand and reproduce system states when servers fail, when specific components need to be replicated, or configuration problems becomes apparent.

The complexity of managing and orchestrating hundreds of virtual servers and application components as well as their dependencies has been addressed by new tools and through shifts in engineering processes in recent years. Important tools for managing large virtual infrastructures include cloud operating systems, resource orchestrators, and automation engines. At the same time, multiple previously mostly distinct software development roles are increasingly combined into new roles under the umbrella of *DevOps* and its ongoing paradigm shift. Moreover, describing desired cloud infrastructures in source code, aptly referred to as *infrastructure as code (IaC)*, which can be executed to recreate systems with the help of automation engines, makes it significantly easier to reproduce defined cloud system states.

Figure 2 depicts infrastructure code alongside the central systems for managing cluster resources.

3.1 Tools for Cloud Resource Management

Cloud operating systems, container orchestrators, and automation tools help to manage large infrastructures and system deployments based on declarative descriptions of the desired state of servers, networks, and systems. These three classes of systems are often used together and can collectively make otherwise manual and often not well documented administrative processes so repeatable that it becomes normal to re-build infrastructures and systems regularly, instead of making changes to already deployed and running systems.

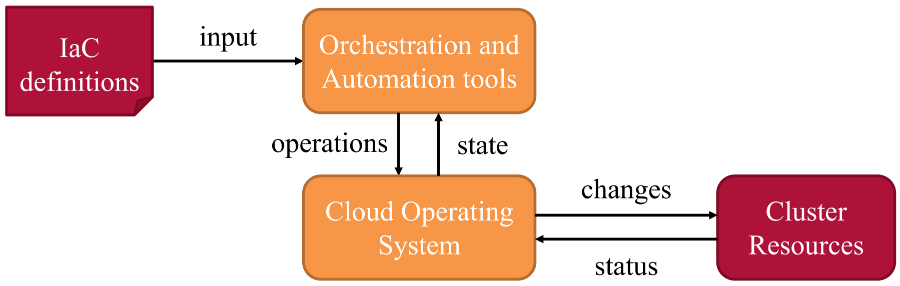

Figure 2: Infrastructure definition code is the input for orchestration and automation tools, which translate usually declarative specifications of desired cluster states into specific imperative operations for cloud operating systems managing infrastructures.

3.1.1 Cloud Operating Systems

Cloud operating systems like OpenNebula[6] [83] and OpenStack[7] [105] allow to provision sets of virtual resources that run on physical infrastructures of data centers. For this, they provide dashboards and APIs, so that resource provisioning can be done by both users and systems.

OpenStack is an open-source cloud operating system mostly deployed to manage IaaS clouds and provide virtual machines. It can, however, also provide access to containers and bare-metal resources. It is a modular systems build around core components for compute resources, images, block storage, and virtual networking. On top, OpenStack provides APIs for programmatic use as well as dashboards for users and administrators. Advanced features include auto-scaling, load balancing, and resource orchestration.

3.1.2 Resource Orchestration Tools

Orchestration systems are control systems for managing clusters of resources. Their main tasks are scheduling and deployment, network and storage configuration, monitoring and logging, as well as component replication. Advanced features of orchestration systems include auto-scaling and load balancing. Typically, orchestration tools take declarative descriptions of the cluster state as input and translate these into imperative operations. Then they continuously monitor cluster states to trigger operations when clusters deviate from the states specified by users, for instance, after component failures or with changing loads.

A prominent example for an orchestration system for Linux containers is Kubernetes[8] [21]. It manages cluster nodes and groups of containers. These container groups, called *pods*, belong to one application and are deployed on one cluster node. Usually these pods run replicated, are scheduled across different nodes, and the load is balanced by Kubernetes. More advanced features of the orchestrator include horizontal auto-scaling and possibilities to influence its scheduling.

[6] `https://opennebula.io/`,accessed 2022-02-16
[7] `https://www.openstack.org/`, accessed 2022-02-16
[8] `https://kubernetes.io/`, accessed 2022-02-16

3.1.3 Automation Engines

Automation engines like Ansible[9], Chef[10], and Puppet[11] work similarly to many orchestration systems. They also translate declarative specifications into imperative operations that move infrastructures and systems to a desired state and then continuously monitor and, if necessary, re-establish the state. In contrast to orchestration tools like Kubernetes, however, their scope is much bigger, making them more flexible tools. A tool like Ansible, for instance, can be used for everything from configuring access rights to installing dependencies on a group of servers as well as to starting and stopping the services of a particular user.

Ansible both defines a language and is an engine for automating configuration management and resource orchestration. The language is declarative, so users describe the state they want systems to be in, not the steps necessary to get there. The system design is agent-less, so it connects to components as a user would, relying on ssh connections instead of implementing a continuously running process for various platforms. Moreover, Ansible provides ways to describe specific tasks, set variables, define hosts, group these different entities, and finally make use of this to manage server resources and configurations.

3.2 Practices Relevant for Building Cloud Infrastructures

Configuration drift and snowflake servers are introduced when virtual IaaS resources, operating systems and middlewares, or distributed applications components are installed, configured, updated, or otherwise maintained without proper documentation and full reproducibility. The main idea addressing these problems is always establishing desired states of infrastructures completely from basic virtual resources, in contrast of applying any partial changes to already running components. Moreover, the build process should be described in a declarative manner, expressing the desired state, not the steps to reach it and manual steps should be avoided in this process. When infrastructures are managed in this way —– constantly rebuilding them to implement adjustments —– the infrastructures are referred to as *immutable infrastructures*.

A closely related idea is continuous integration/delivery (CI/CD). CI/CD aims to provide faster feedback by integrating smaller changes and having these reach production systems sooner. The practice advocates constantly building, testing, and releasing small sets of changes, instead of having individual developers gather larger sets of changes locally or waiting to release bigger adjustments to customers less frequently. The reason for this is that developers get feedback faster. Developers might notice a problem with a specific code change quicker in production right after they made the change, while still very aware of all the reasoning around the change. This approach ties nicely with the idea of combining roles in DevOps, as it

[9]`https://www.ansible.com/`, accessed 2022-02-16
[10]`https://www.chef.io/`, accessed 2022-02-16
[11]`https://puppet.com/`, accessed 2022-02-16

is then the same person or team noticing the issue in production and responsible for coming up with a fix for the issue.

3.2.1 Infrastructure-as-Code

The central idea of IaC [85] is to provide full reproducibility and increased documentation by capturing every aspect of the server setup in source code that can be versioned and shared among users. This infrastructure code is then executed to re-build the entire infrastructure and parts of services to make changes. Typically, the infrastructure code declaratively describes the desired cluster state and runs on automation engines.

The key is to default to re-building everything from scratch instead of actively updating and maintaining already deployed infrastructure and application components. If this is done for every change, infrastructures do effectively become immutable, once deployed. With this approach, virtual infrastructures become disposable as they can be re-build efficiently at any time, so that cluster states become reproducible. Moreover, the configuration and management steps needed to establish specific cluster states are also documented by the infrastructure code. The declarative code typically visibly distinguishes between deliberate and default configuration as well as between specific hosts and groups of hosts, thereby reducing the complexity of the management as the entire process becomes more transparent to everyone involved.

3.2.2 DevOps and Continuous Delivery

The DevOps movement integrated tools and processes from previously more distinct roles in software development, quality assurance, and system operations [10]. People responsible for specific components or microservices, often referred to as *DevOps Engineers*, are responsible for implementing the specific application part, for testing it, and for making sure it runs correctly in the production environment. This combination of responsibilities aims to foster faster feedback as well as to have smaller increments of working software reach users quicker.

The DevOps paradigm is rooted in the ideas of iterative software development. The idea is that iterations between code, test, and deploy can be shorter without handovers between distinct teams between development, testing, and deployment. Moreover, if a person or team runs tests directly after making changes to a system, feedback loops are short. The same is true when the responsibilities also include monitoring application parts in production, where there is always the chance that real-world usage differs from implemented test cases. With the same people responsible for developing and keeping a component running, they are likely to be able to spot and fix issues with any new versions they deployed quickly, simply because of the familiarity with the specific change and application part in general. In contrast, when a development team instead hands over new versions to a testing team, which then in turn forwards tested updates to an operations team, there is knowledge lost with each handover.

A central practice of the DevOps paradigm is to continuously integrate, test, and deploy software. Instead of integrating concurrent developments with a main version of the software only sporadically, making it potentially difficult to integrate diverging changes, developers merge their changes into a main version continuously. So, whenever they finish an atomic change, moving a system's implementation from one version that can be build and deployed to another, they push their changes to a common repository. Thus, teams have a common view on the state of their software project, only different by small changes currently being worked on. Moreover, teams then use tools that automatically build the software and run the implemented test cases against it. These tools are referred to as *CI servers*. These server tools react on every change that is pushed to the mainlines of software projects, pull the latest version, build it, and test it. This practice is called *continuous integration (CI)*. CI servers contribute to the common view on the state of software projects as they prominently display the current state of the mainlines of a central repository. Typically, they show states as *red* and notify all responsible engineers whenever a specific version does not build and test as expected. Then further changes should not be pushed, before the current issue is fixed and the system is in a *green* state again. Beyond CI, there is *continuous delivery (CD)*, which also builds and tests code, yet then furthermore automatically deploys new versions of the software, at least to staging environments – but possibly also to production, if no issues are detected while new versions run on the staging environments.

4 Scalable and Fault-Tolerant Distributed Cloud Applications

Virtualization and IaaS clouds make it easy to provision large clusters of resources for users. However, scaling applications effectively to many resources requires that components are replicated and that the load among these replicated components is well balanced.

Both scalability and fault tolerance are addressed by horizontal scaling, in which components are replicated across an increased number of resources, in contrast to vertical scaling, in which components are instead deployed to the same number of resources yet ones with increased capacities. Scaling, deployment, monitoring, and re-starting of failed components is supported by orchestration and automation tools. Such tools furthermore often support auto-scaling. The decision to scale in or out can be triggered by simple metrics and thresholds such as resource utilization averages, by sets of rules supplied to automation engines, or by models of the scale-out behavior.

An architectural pattern well aligned with the goal of scalability and fault tolerance are microservice architectures, in which distributed applications are build out of small components that each can be developed, tested, deployed, scaled, and recovered separately.

4.1 Stateless Components

Stateless components only require replication and load balancing for scalability and availability even in case of failures. That is, since no state is managed beyond the execution scope of a single request or invocation, these components neither need to partition nor to replicate any state across workers. This makes it easy to increase and decrease the number of deployed stateless components dynamically in response to loads and while monitoring defined QoS requirements.

Load balancing between stateless applications can be implemented efficiently using network protocols such as Internet Protocol (IP) and Domain Name System (DNS), as long as requests fit into a single packet. IP anycast messages are forwarded automatically to one of multiple receivers that share an IP address, while the DNS protocol was designed around the idea of an hierarchical load balancing, where local and regional DNS servers can resolve domain names to different IPs and, therefore, servers. This is used with many global services such as search engines, web applications, news portals, and streaming services, where globally distributed server networks serve the same data to users in their proximity.

4.2 Stateful Components

Stateful components on the other hand use data partitioning for scalability or, additionally, data replication for both scalability and fault tolerance. Partitioning is typically done horizontally using partitioning functions like hash partitioning or range partitioning. Horizontal partitioning, which distributes the same type of data across workers, is called sharding. If data is replicated in a distributed system, this distributed state can become inconsistent. Developers and users need to be aware of the level of provided consistency. Therefore, distributed systems typically guarantee a specific level of provided consistency. But, consistency models have an impact on the scalability of applications. Stronger models include strict consistency, linearizability, and sequential consistency. A model popular with many scalable databases such as NoSQL data stores is eventual consistency [65]. The only guarantee provided here is that all replicas will converge to the same state at some point in time.

A prominent research result regarding scalability and consistency is the CAP theorem [20]: Essentially, given commodity hardware and large-scale systems, network components will fail resulting in a partitioned network, in which certain components cannot reach specific other components. For this situation, system architects need to decide beforehand between consistency or availability: when the data on reachable replicas can still be changed unreachable replicas become temporarily inconsistent or, else, even the reachable components can no longer be updated to preserve consistency with the unreachable components. That is, the system is for the time of the network partitions either inconsistent or unavailable.

Besides often not providing strict consistency, NoSQL data stores typically sacrifice consistency in case of network partitions [99]. This design makes the data stores more scalable, especially horizontally across cluster resources such as a set of VMs in a public cloud or a provider's PaaS data storage service. Data in NoSQL stores is often organized in some form of tables, yet is designed to accommodate various types of data such as graphs, documents, key-value data, and time-series data [99]. Though, while many of these data stores still apply the notion of tables, rows, and column, they frequently do not require upfront table schemas, as is, for instance, the case with many wide-column stores. Furthermore, many NoSQL stores provide other programmatic interfaces than SQL such as low-level query interfaces, which often do not follow a defined standard.

4.3 Microservices Architectures

Server applications typically consist of multiple components. Usually there are at least different components for the user interface, the business logic, and storing data. In a microservice architecture [26], though, system are broken up in many more services that can be deployed and scaled individually, with advantages for their development, testing, and operation, yet often also introducing new challenges. Figure 3 contrasts a microservice architecture to a monolithic system.

A good practice is to design systems so they consist of components that each exhibit high levels of cohesion yet low levels of coupling. This can be achieved by breaking up monoliths into many small and mostly isolated components in a microservice architecture, yet certainly still relies on carefully designing individual components, so that they can be understood, adapted, and put to use without many dependencies.

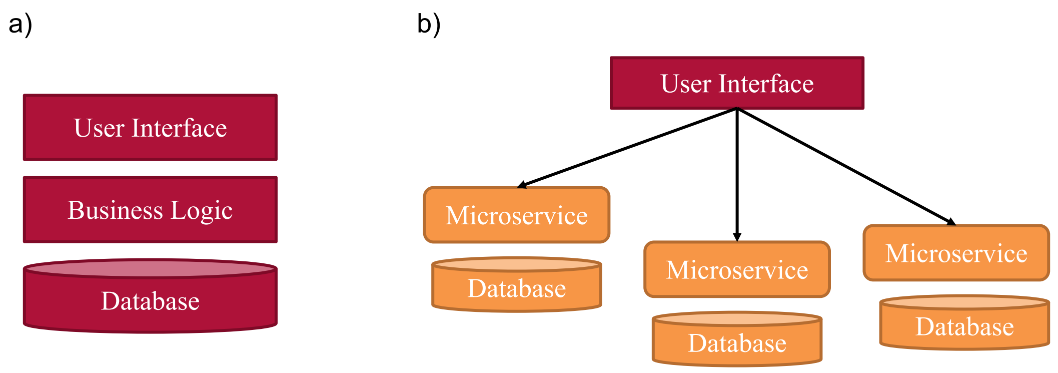

Figure 3: Architecture styles for multi-component cloud applications: a) a classical three-tier application design (monolith), contrasted to b) a microservice architecture.

Microservices can be developed by separate teams that are typically free to use the technologies of their choice for implementing the service. Moreover, each of the services can be deployed, replicated, and scaled individually. This, however, does arguably lead to system designs that consist of many components that each need to be configured, managed, and monitored, often in a range of different technologies, so that each component is best understood and advanced by the responsible team.

Generally, microservices allow for a much more fine-grained orchestration (i.e. configuration and dependency management, monitoring and failure handling, replication and load balancing on the basis of small components that in the best case exhibit strong cohesion and loose coupling), yet keeping track and managing many small services can also quickly become difficult for large-scale microservice architectures.

5 Data-Intensive Distributed Cloud Applications

Increasing amounts of data are being recorded and generated. A major reason for this is the digitalization and the IoT. More business and other processes are automated with information and communications technology. Large numbers of sensors deployed in cities, factories, and homes continuously record and emit data. A growing number of sciences generates large volumes of data in experiments. User-generated content, especially in connection with the Web and the enormous user bases of Internet companies, is another major reason for the increasingly large volumes of data. Additional, users also generate immense datasets in the form of log data, which companies process for their core business but also to gain further insights into their customers.

The increasing size of datasets, the decreasing prices of commodity hardware, in particular for long-time storage, and the possibilities of leasing as many resources as needed in clouds have led to the development of distributed systems that utilize clusters of homogeneous resources for handling large amounts of data. The two main tasks here are (i) providing access to and (ii) facilitating the analysis of large volumes of data.

5.1 Distributed Data Storage

The main classes of systems that provide access to large amounts of data are distributed storage systems and scalable data stores. These systems distribute data by splitting up larger files into blocks, also called chunks, or tables into partitions, which are then distributed across a set of workers. Moreover, typically data is replicated for fault tolerance and potentially also scalability, using a load balancer. This approach is exemplified in Figure 4, in which a file is split up into multiple blocks, which are then stored on a set of nodes, with each block being stored twice in the cluster. There are several scalable data stores that directly build on top of distributed storage systems, for instance NoSQL databases that make use of underlying distributed file systems.

Prominent examples for distributed file systems are the Google File System (GFS) [39] and Hadoop Distributed File System (HDFS) [106]. GFS stores large files across commodity cluster, following a main-worker architecture and storing chunks of large files replicated across workers. HDFS is an open-source implementation of the same design. Ceph [125] is also a popular distributed storage system. In

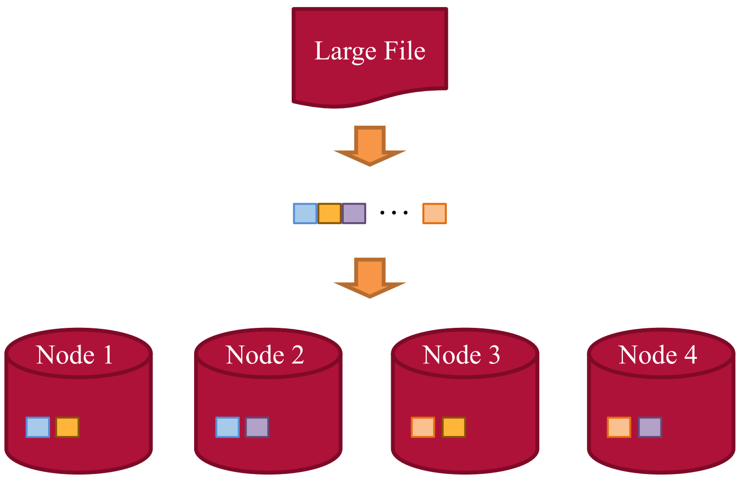

Figure 4: A large file is split up into multiple blocks and replicas of these blocks are stored across a set of connected worker nodes.

addition to file storage, it also stores objects for applications and block devices to VMs.

Distributed systems for managing more structured data include the wide-column stores BigTable [24] and Cassandra [68]. Without requiring an upfront schema, these systems store data in rows and columns, allowing specific parts of the data to be indexed and read efficiently. They also gather writes in memory before committing them to disks, secured by logs, to provide fast, yet dependable write access. The systems replicate and partition data, with eventual consistent replicated data points and the data being available even in case of network issues within a cluster.

5.2 Distributed Data Processing

Distributed systems for the analysis of large volumes of data usually implement data-parallelism. An important class of distributed data-parallel processing systems are those following a dataflow model such as MapReduce [32], SCOPE [23], Spark [132], and Flink [22]. These systems typically offer restricted programming models that allow users to construct programs by composing graphs of operators. These operators are second-order functions such as Map and Reduce, yet also typical database operators like Joins. The execution models and runtimes of these systems then execute the operators of a job graph with specific levels of data-parallelism. That is, multiple task instances of operators in a job graph process data at runtime. Each task instance handles a partition of the entire data flowing through the job. Figure 5 shows an example dataflow graph to be submitted in parallel on a set of cluster nodes.

The systems take care of task parallelization, data partitioning, scheduling and deployment, communication between nodes, and fault tolerance. Fault tolerance is usually achieved by checkpointing intermediate data or by keeping track of linage, to be able to re-compute specific lost partitions. This way, dataflow systems support

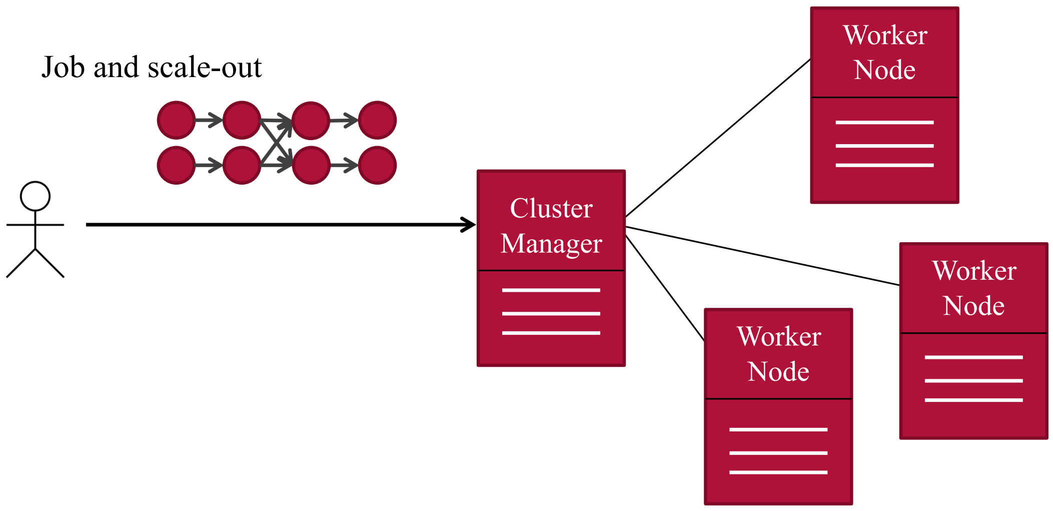

Figure 5: A distributed dataflow job is submitted by a user to a cluster manager in order to run the job in parallel on a set of worker nodes.

users by providing high-level programming abstractions that hide the complexities of parallel programming and distributed systems, allowing users to express distributed data-parallel programs by writing sequential code only, while the system's distributed runtimes realize an efficient and fault-tolerant distributed execution of these programs, at least as long as problems fit the restricted programming and execution models well enough. The problems that do fit well are data-parallel to a large extent, where the amount of parallelizable work exceeds the overhead introduced by distributed communication.

For continuous monitoring, be it to oversee large IT infrastructures or the physical world with the distributed sensors and devices of the IoT, systems need to be able to ingest and process data streams. For these use cases existing batch processing systems such as Spark and Flink have been equipped with support for stream processing. At the same time, there are also dedicated stream processing systems such as Storm [119] and Samza [89]. Essential features for distributed stream processing include windowed aggregations, mechanisms for out-of-order processing, and fault tolerance mechanisms for continuous streams, usually based on checkpointing input data and operator state, besides the general requirement of low-latency processing of the continuously arriving data. Typical fault tolerance requirements are at-least-once and exactly-once processing. Two principle strategies for continuous low-latency processing are using small batches that match the required windows of a program, which is referred to as microbatching and was used for Spark Streaming [133], and actual continuous processing with permanently deployed stream processing jobs as done for instance by Storm and Flink. Typically, the latter can achieve lower latencies, yet this often comes at the expense of throughput. Both Flink and Spark offer abstractions to realize batch and streaming jobs, thereby providing unified data processing platforms. Similarly, Google's Dataflow model [1], with a managed service to run Dataflow jobs on Google's cloud, as well as the Apache Beam project[12] to run jobs on open-source systems such as Flink, also execute both batch processing and streaming jobs.

Systems like MapReduce, Spark, and Flink have been used for relational data processing [5, 38, 90, 117, 131], large-scale graph analysis [44], and machine learning [72, 81]. Though inspired by dataflow systems, new distributed systems have been designed and implemented for deep learning, most notably Tensorflow [92] and PyTorch [92]. These systems are optimized to compute the gradient of a given loss function for large training datasets, implementing parallel training on CPUs and GPUs as well as clusters and other resources.

Using large sets of virtual resources such as VMs or containers for running data-parallel systems offers advantages especially to organizations and users with irregular or growing large-scale data processing needs. Today there are also various fully managed platforms like Amazon EMR[13], which can be used to run distributed data-parallel processing systems such as MapReduce and Spark without managing cluster infrastructures and systems.

6 Cloud Platforms and Managed Services

Managed cloud platforms provide their services at a higher level of abstraction than infrastructure services, yet in comparison to the SaaS model they still allow consumers to deploy their own programs using the platform's programming model, libraries and runtime, and infrastructure. That is, consumers do not control the operating system, server hardware, network, and storage, but they still can run their own applications as long as those fit the platform's programming system.

6.1 Platform Services

A platform's programming system is made up of pre-defined programming languages and models, available libraries and means for interactions with external systems, as well as possibly development tools. The platform providers then take care of the operation and maintenance of the entire hardware, operating system, and middleware layers. Platforms often provide integrations with other platforms by the same provider, like compute platforms that are well integrated with managed storage and web servers. Their task is furthermore to ensure availability in case of planned maintenance and even unforeseen outages as well as enabling transparent scalability. This way consumers have a significantly reduced development and maintenance effort, yet this comes at the cost of flexibility as applications often need to be written for a particular platform, using the specific APIs of this platform, and also slightly higher costs than equivalent resource usage on the IaaS level.

[12]`https://beam.apache.org/`, accessed 2022-02-16
[13]`https://aws.amazon.com/emr/`, accessed 2022-02-16

Platforms typically charge on a much finer granularity than IaaS services. That is, not just by the time of usage, but per query, message, request or similar fine-grained metrics. This allows customers to reduce upfront expenses even more and subsequently applications deployed on platforms can have very low operational costs until they become popular.

6.2 Serverless Computing

Serverless computing is an execution model in which customers do not need to manage resources since the services scale dynamically automatically. This model is enabled by high-level PaaS offerings, namely function as a service (FaaS) and backend as a service (BaaS).

At an even higher level than the initial PaaS platform offerings, many cloud providers today also offer services in FaaS service models, in which consumers only write single functions that are executed in a fully managed environment as events or requests are received. That is, the functions are event-triggered and ephemeral. They are required to be stateless and there is typically a maximum execution time for each function invocation. Given the reduced scope of programs on FaaS platforms, they align nicely with microservice architectures [7].

FaaS platforms start and scale function instances horizontally as necessary. They do not induce any costs as long as no events or requests trigger functions to be executed, shifting costs from reservations to paying only for actual usage.

Besides relying on FaaS components, serverless computing architectures usually also make use of BaaS offers. BaaS are fully managed services for specific backend functions such as user authentication, session state, and databases. Firebase was an early provider of BaaS offerings that included a database and the synchronization of session state. AWS Cognito is an example of an BaaS service for user authentication.

While serverless computing's comprehensive resource management and ease of use have led to considerable adoption, there are also voices criticizing the simplistic auto-scaling and scheduling policies implemented with many FaaS platforms today, which are arguably at odds with the data-driven, distributed computing models that have been developed and became popular over the last two decades [52]. In particular, decoupling computation and storage can quickly lead to poor data locality, if data first has to be moved to functions, before they can execute. Similarly, if systems only allow for isolated, ephemeral, and stateless functions, without supporting any composition of programs out of individual functions, passing results from one function to the next can also lead to inefficiencies. For this reason, several research and development efforts aim to provide efficient implementations of addressable stateful functions. For instance, Apache Flink StateFun[14] implements an API and runtime for stateful functions on top of the Apache Flink system. Another example is Cloudburst/Anna [110, 130]. Like StateFun, this project addresses the shortcomings of current serverless FaaS platforms by providing stateful functions.

[14]`https://github.com/apache/flink-statefun`, accessed 2022-02-16

It does so by relying on the Anna autoscaling key-value store for state sharing and mutable caches co-located with function executors for data locality, thereby implementing a logical separation of compute and storage, yet at the same time a physical co-location.

A concern raised with all managed platforms and typically increasingly with higher levels of abstraction is vendor lock-in, since programs then rely considerably on the particular environment of specific platforms.

7 Fog Computing and the Internet of Things

With a growing number of connected IoT devices and sensor platforms that are widely distributed across people's homes, cities, and factories [2, 29, 58, 60], there is a need for system architectures that can record and process the data generated by these devices.

Fog computing is a conceptual model that tries to create architectures that allow for widely distributed systems in which local fog nodes can react quickly to sensor inputs. As such, fog computing moves some of the characteristics of cloud computing closer to traditional control systems.

QoS requirements for some applications are increasing as more and more use cases require response times within pre-defined latency bounds. As networks are not yet capable to reliably satisfy these requirements, shortening the distance between customers and cloud resources—and therefore placing aspects of the processing closer to the sources of data—is the currently followed approach addressing this need for bounded latencies.

For example, in urban infrastructures like public transport systems or water networks more and more sensors get deployed, which then continuously record and emit environment information that are processed to have infrastructures quickly adapt to measured environment conditions automatically.

Typically, the sensor data streams are received, handled, and forwarded by sensor platforms, which are small devices. These devices are installed at the source of the sensor data and widely distributed across the infrastructures. They are connected to gateways, for instance using wireless communication technology for sending to and receiving from mobile access points. Often, the data ultimately ends up at central clusters of cloud resources, where a perceived unlimited amount of compute and storage resources are available. However, before reaching these central cloud resources the sensor data streams first go through a number of edge/fog layers, each with increasing amounts of available resources, but each also adding more latency and typically aggregating streams from larger numbers of sensor platforms, as shown in Figure 6.

It can offer significant advantages in terms of reduced latencies and saved bandwidth as well as overall system scalability to deploy computation tasks directly on IoT devices and other edge/fog layers close to the data sources. The idea of the fog and edge computing paradigms is therefore to utilize not only central cloud resources, but to extend execution models beyond clouds and thereby utilize the

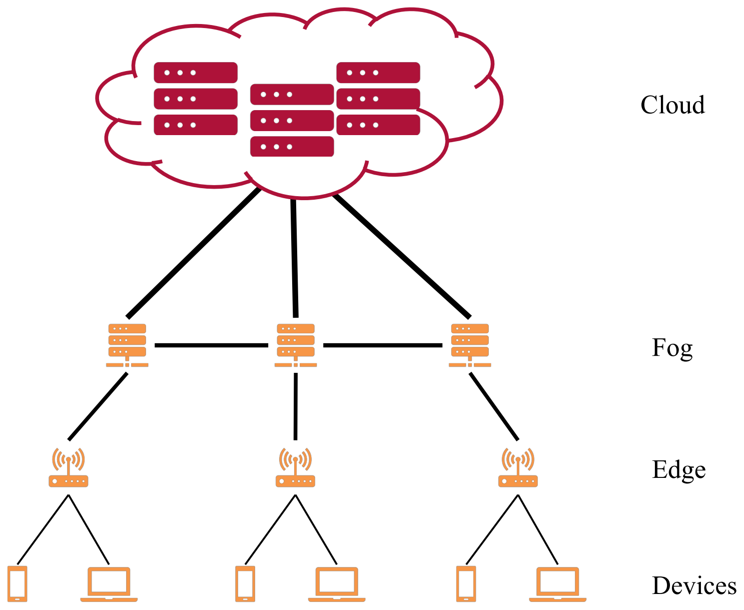

Figure 6: Available resources in IoT environments: from distributed devices over local access points and intermediate resources to finally abundant virtual resources in clouds.

available resources in-between the sensor sources and cloud resources. At the same time, task execution on IoT devices and other edge resources might not be feasible for more compute-intensive tasks as well as for storing large amounts of data due to the often significantly constrained capacities at these layers, especially in case of fully autonomous, battery-powered sensor platforms.

This leads to a pattern in many fog architectures, where edge and fog nodes on the lower levels are mainly executing simpler but very data-reducing tasks like averaging data in time windows and are responsible for latency-sensitive computations similar to traditional control systems. The more resource-intensive tasks like data analytics are then done in the upper levels and the cloud.

8 Research Outlook

Extensive ongoing research in the methods of cloud computing aims to further automate operational tasks such as cloud system monitoring and resource configuration. In addition, the advent of edge and fog computing poses a multitude of new challenges, which need to be addressed to further decentralize cloud infrastructures.

8.1 Increasing Cloud Automation

Given the scales of today's infrastructures, the large numbers of replicated components that cooperatively respond to the requests of thousands of users in many cloud applications, and the high number of concurrent tasks used in massively parallel data processing, further automation of IT operations is absolutely essential. Human operators are simply no longer able to grasp the sheer amount of monitoring and log information at these scales. A major current trend in addressing this problem is ML-supported IT operations, in which extensive monitoring data is used with machine learning methods in order to provide as-automated-as-possible operations based on the current measured state of systems. This is applied for various resource management problems, including for capacity planning, load balancing, scheduling, threat analysis, and anomaly detection [43, 46, 55, 67, 75, 84, 88, 115, 118, 134]. Given the amount of continuous monitoring data and the possibility of concept drifts, especially semi-supervised and unsupervised machine learning methods such as clustering algorithms lend themselves to analyze monitoring data on-the-fly and with little human involvement [56, 103, 107]. On the other hand, with recurring tasks such as periodically scheduled batch jobs also supervised methods can be applied, as previous executions can be used to train models for future executions.

To ensure a high availability and provide resilient clouds a central goal for distributed applications and services is noticing and increasingly also remediating anomalies and threats before they become failures, so downtimes of systems can be avoided or at least reduced [30, 78, 79]. This is especially necessary in light of the continued trends of using commercial off-the-shelf hardware and open-source software. For instance, even the critical services of telecommunication service providers, which are known for their high levels of required availability, have been migrated to open-source software stacks and virtualized network functions [9, 45, 50]. Such use cases demand to move beyond reacting to failures and instead require anticipating failures, often using machine learning.

To ensure a high degree of efficiency and performance of distributed data-parallel processing jobs, a central approach is modeling and predicting the scale-out behavior and the impact of heterogeneous resources, data locality, and interference among co-located workloads [3, 6, 33, 54, 62, 114, 116, 122, 123]. That is, using statistical models, resource management systems automatically select adequate types and amounts of cloud resources for the performance requirements of jobs, schedule applications with complementary resource demands onto shared resources, and continuously adapt reservations to workload characteristics monitored at runtime. With this, systems aim to provide the required performance and dependability, yet at the same time also utilize resources well.

An optimization objective that warrants special attention is the goal to reduce a workload's footprint on the environment through the associated greenhouse gas emissions of consumed energies. Despite steadily improving efficiencies in large-scale centralized cloud data centers, energy demands are expected to further increase in the next years [76]. One approach to this problem are efforts to shift work to when and where energy supplies have a relative low carbon-intensity [47, 96, 127, 137], because the energy is supplied from renewable sources such as solar or wind, relying on machine learning to forecast as well as match supply and demand.

A recurring challenge when using machine learning methods to optimize the monitoring and management of cloud infrastructures is the unavailability of training data for applications and jobs to be executed. One specific trend addressing this issue is to use reinforcement learning, so systems continuously learn, for instance, how to schedule workloads based on a function expressing the quality of scheduling decisions [28, 73, 74, 98, 104, 113, 136]. Another approach is to share and reuse runtime data and possibly also performance models across different execution contexts such as specific clusters [101, 102, 129], assuming that in the end many people run the same algorithms, often using implementations stemming from popular libraries, and also the same kind of resources, such as in public clouds.

8.2 Increasing Cloud Decentralization

The Internet of Things, cloud computing, and machine learning will allow for more adaptive cities, factories, and houses, yet this vision of intelligent cyber-physical systems will not be implemented with centralized cloud resources alone. Instead, new distributed computing paradigms for the IoT such as edge and fog computing will bring resources closer to sensor-equipped edge devices [31]. This will have considerable advantages, including lower latencies, reduced network bottlenecks, decreased energy consumption for wide-area communication, and in many cases also considerable benefits for security and privacy. At the same time, this shift — away from centralized large-scale cloud data centers of a few major provides and towards decentralized resources of many more and also smaller providers — will come with new challenges. The emerging compute infrastructures, spanning a continuum from IoT devices and edge resources to clouds, will be significantly more heterogeneous, dynamic, and distributed. First, resources of different providers, with various capacities and capabilities, will need to communicate and cooperate. Second, resources will also far more frequently join and leave as well as move around. Third, resources will be distributed across larger distances, relying on wide-area networks and a variety of network protocols and protocol implementations.

8.2.1 Resource Management

A key challenge with the shared compute infrastructures of the future will be the efficient use of all resources, including for compute, storage, and networking. This is currently addressed at various levels of abstractions, yet often presents itself as a problem of allocation and scheduling [40].

In decentralized architectures, without perfect knowledge of globally available resources and workloads, the resource management consequently needs to use locally available information to schedule tasks and allocate resources. One possibility is the opportunistic information exchange between, and subsequent task offloading to, nearby peers in edge and fog computing [11, 69].

Embedding the task scheduling into well-known programming paradigms sometimes allows for better placement decisions. The resource management in fog architectures has consequently been investigated in the context of stream processing systems and in the serverless computing model, among others. When adapting the concept of distributed stream processing systems to the execution on distributed heterogeneous nodes [53, 57, 86, 135], the scheduling needs to take current resource capacities of dynamically changing distributed compute and network infrastructures into account. In distributed stream processing, the scheduling decision itself is still usually controlled centrally. In the context of the serverless computing model, there are concepts to schedule the function executions in FaaS frameworks on edge and fog nodes [87, 93, 97], taking the heterogeneity of the execution nodes into account. The placement of the functions executions often presents itself as an offloading decision that is triggered by reaching resource limitations or in some cases efficiency and energy consumption considerations.

8.2.2 Machine Learning

The increasing decentralization of previously typically centralized systems can also be seen in the area of machine learning, where the desire to improve machine learning on private data led to the concept of federated learning. In federated learning multiple decentralized clients can train a common machine learning model on their private local data without the data leaving devices but rather by transferring the updates of the weights of a neural network. These weight updates are sent to a central server that creates a new common model by averaging the updates, which is sent back to the clients [63, 66].

In cases where there are distinct local characteristics in the data of the distributed clients, personalized federated learning can help to strike the balance between learning global characteristics and specializing on the local data. For this personalization, different methods are discussed, including multi-task learning, meta-learning, and model-agnostic meta-learning [34, 36, 59].

Taking the reduction of central control further, decentralized federated learning replaces the central server with a decentralized consensus mechanism like a blockchain. Thus, not only the data, but also the control of the learning process becomes decentralized [13, 27, 64, 71, 124, 126]. Gossip learning refers to an alternative strategy in decentralized learning where the distributed clients each send their updates to another—most of the time randomly sampled—peer [17, 35, 41, 48, 51, 91, 121].

8.2.3 Testing and Dependability

An important prerequisite for the widespread adoption of IoT systems is that they are tested adequately, especially if they implement critical tasks in areas such as medicine, manufacturing, and cities. However, this is much more difficult for IoT systems, compared to the refined practices and tools available for cloud-based applications.

In software development in general, the continuous testing of small software changes is widely recognized as best practice. For cloud software this is usually facilitated by the use of staging environments in which software changes are deployed automatically through continuous integration / continuous delivery (CI/CD) systems. These staging environments essentially behave like production environments, and—in the practice of blue-green deployment or rolling deployments—become the production environments if no major issue is detected while traffic is gradually routed to the new environments.

While test environments for the software of less centralized architectures like edge and fog computing will not be able to exactly reproduce the production environment to the same extend as this is possible for cloud software, experimentation and continuous testing is possible in test environments using virtual resources and emulation [15, 16, 49, 77, 111, 128].

When using such test environments for IoT software, one important tradeoff happens between the realism and the scale of the test environment. Hardware testbeds allow for more realistic testing, while the testing anywhere close to an actual scale is usually only possible with virtual test environments. Hybrid test environments try to incorporate both virtual and physical nodes to make this tradeoff simpler and get some benefits from both approaches [12, 18].

Another important aspect of increasing the realism of the test environments of IoT systems is the consideration of not only the IT resources but also the rest of the physical environments in which they will operate. By definition, IoT systems integrate with their environment through actuators and sensors. Including a simulation of the environment—like traffic flow, water network, or energy grid simulations—in the testing of IoT systems can consequently further improve the realism [14, 42].

Acknowledgments

This report is based on the lectures on the topic by the authors at Technische Universität Berlin and Hasso Plattner Institute. We thank the university staff of the past and present with significant contributions to the course preparation, delivery, and organization: Daniel Warneke, Björn Lohrmann, Mareike Höger, Anton Gulenko, Sven Köhler, Lukas Pirl, Morgan Geldenhuys, Ilja Behnke, and Alexander Acker.

License and Availability

References

[1] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. "The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing". In: *Proc. VLDB Endow.* 8.12 (2015).

[2] A. Al-Ali, I. A. Zualkernan, M. Rashid, R. Gupta, and M. Alikarar. "A Smart Home Energy Management System Using IoT and Big Data Analytics Approach". In: *IEEE Trans. Consumer Electron.* 63.4 (2017).

[3] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang. "Cherrypick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics". In: *NSDI'17*. 2017.

[4] M. Armbrust, I. Stoica, M. Zaharia, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, and A. Rabkin. "A View of Cloud Computing". In: *Commun. ACM* 53.4 (2010).

[5] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. "Spark SQL: Relational Data Processing in Spark". In: *SIGMOD'15*. ACM, 2015.

[6] J. Bader, L. Thamsen, S. Kulagina, J. Will, H. Meyerhenke, and O. Kao. "Tarema: Adaptive Resource Allocation for Scalable Scientific Workflows in Heterogeneous Clusters". In: *BigData'21*. IEEE, 2021.

[7] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter. "Serverless Computing: Current Trends and Open Problems". In: *Research Advances in Cloud Computing*. Springer, 2017.

[8] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. "Xen and the Art of Virtualization". In: *SOSP'03*. ACM, 2003.

[9] H. Basilier, M. Darula, and J. Wilke. "Virtualizing Network Services–the Telecom Cloud". In: *Ericsson Review* 91 (2014).

[10] L. Bass, I. Weber, and L. Zhu. *DevOps: A Software Architect's Perspective*. Addison-Wesley, 2015.

[11] S. Becker, F. Schmidt, L. Thamsen, A. J. Ferrer, and O. Kao. "LOS: Local-Optimistic Scheduling of Periodic Model Training For Anomaly Detection on Sensor Data Streams in Meshed Edge Networks". In: *ACSOS'21*. IEEE, 2021.

[12] I. Behnke, L. Thamsen, and O. Kao. "Héctor: A Framework for Testing IoT Applications Across Heterogeneous Edge and Cloud Testbeds". In: *UCC'19*. IEEE, 2019.

[13] J. Beilharz, B. Pfitzner, R. Schmid, P. Geppert, B. Arnrich, and A. Polze. “Implicit Model Specialization through DAG-based Decentralized Federated Learning”. In: *Middleware'21*. ACM, 2021.

[14] J. Beilharz, P. Wiesner, A. Boockmeyer, F. Brokhausen, I. Behnke, R. Schmid, L. Pirl, and L. Thamsen. “Towards a Staging Environment for the Internet of Things”. In: *PerCom'21*. IEEE. 2021.

[15] J. Beilharz, P. Wiesner, A. Boockmeyer, L. Pirl, D. Friedenberger, F. Brokhausen, I. Behnke, A. Polze, and L. Thamsen. “Continuously Testing Distributed IoT Systems: An Overview of the State of the Art”. In: *ICSOC'21*. Springer, 2021.

[16] F. Bender, J. J. Brune, N. L. Keutel, I. Behnke, and L. Thamsen. “PIERES: A Playground for Network Interrupt Experiments on Real-Time Embedded Systems in the IoT”. In: *ICPE'21*. ACM, 2021.

[17] M. Blot, D. Picard, N. Thome, and M. Cord. “Distributed optimization for deep learning with gossip exchange”. In: *Neurocomputing* (2019).

[18] A. Boockmeyer, J. Beilharz, L. Pirl, and A. Polze. “Hatebefi: Hybrid applications testbed for fault injection”. In: *ISORC'19*. IEEE. 2019.

[19] D. Box and T. Pattison. *Essential .Net: The Common Language Runtime*. Addison-Wesley, 2002.

[20] E. Brewer. “CAP Twelve Years Later: How the "Rules" Have Changed”. In: *Computer* 45.2 (2012).

[21] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. “Borg, Omega, and Kubernetes”. In: *Queue* 14.1 (2016).

[22] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. “Apache Flink: Stream and Batch Processing in a Single Engine”. In: *IEEE Data Engineering Bulletin* 36.4 (2015).

[23] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. “Scope - Easy and Efficient Parallel Processing of Massive Data Sets”. In: *Proc. VLDB Endow.* 1.2 (2008).

[24] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. “Bigtable - A Distributed Storage System for Structured Data”. In: *ACM Trans. Comput. Syst.* 26.2 (2008).

[25] C. Chaubal. “The Architecture of VMware ESXi”. In: *VMware White Paper* 1.7 (2008).

[26] L. Chen. “Microservices: Architecting for Continuous Delivery and DevOps”. In: *ICSA'18*. IEEE, 2018.

[27] X. Chen, J. Ji, C. Luo, W. Liao, and P. Li. “When Machine Learning Meets Blockchain: A Decentralized, Privacy-Preserving and Secure Design”. In: *IEEE BigData'18*. 2018.

[28] M. Cheong, H. Lee, I. Yeom, and H. Woo. “SCARL: Attentive Reinforcement Learning-Based Scheduling in a Multi-Resource Heterogeneous Cluster”. In: *IEEE Access* 7 (2019).

[29] M. Chiang and T. Zhang. “Fog and IoT: An Overview of Research Opportunities”. In: *IEEE Internet Things J.* 3.6 (2016).

[30] Y. Dai, Y. Xiang, and G. Zhang. “Self-Healing and Hybrid Diagnosis in Cloud Computing”. In: *Lecture Notes in Computer Science*. Springer, 2009.

[31] A. V. Dastjerdi and R. Buyya. “Fog Computing: Helping the Internet of Things Realize Its Potential”. In: *Computer* 49.8 (2016).

[32] J. Dean and S. Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. In: (2004).

[33] C. Delimitrou and C. Kozyrakis. “Quasar - Resource-Efficient and QoS-Aware Cluster Management”. In: *ASPLOS'14*. ACM, 2014.

[34] Y. Deng, M. M. Kamani, and M. Mahdavi. “Adaptive Personalized Federated Learning”. In: *arXiv preprint arXiv:2003.13461* (2020).

[35] M. A. Dinani, A. Holzer, H. Nguyen, M. A. Marsan, and G. Rizzo. “Gossip Learning of Personalized Models for Vehicle Trajectory Prediction”. In: *WCNCW'21*. 2021.

[36] A. Fallah, A. Mokhtari, and A. Ozdaglar. “Personalized Federated Learning: A Meta-Learning Approach”. In: *arXiv preprint arXiv:2002.07948* (2020).

[37] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. “An Updated Performance Comparison of Virtual Machines and Linux Containers”. In: *ISPASS'15*. IEEE, 2015.

[38] E. Friedman, P. Pawlowski, and J. Cieslewicz. “SQL/MapReduce: A Practical Approach to Self-describing, Polymorphic, and Parallelizable User-defined Functions”. In: *Proc. VLDB Endow.* 2.2 (2009).

[39] S. Ghemawat, H. Gobioff, and S.-T. Leung. “The Google File System”. In: *SOSP'03*. ACM, 2003.

[40] M. Ghobaei-Arani, A. Souri, and A. A. Rahmanian. “Resource Management Approaches in Fog Computing: A Comprehensive Review”. In: *Journal of Grid Computing* 18.1 (2020).

[41] L. Giaretta and Š. Girdzijauskas. “Gossip Learning: Off the Beaten Path”. In: *Big Data'19*. 2019.

[42] C. Gomes, C. Thule, D. Broman, P. G. Larsen, and H. Vangheluwe. “Co-Simulation: A Survey”. In: *ACM CSUR* (2018).

[43] K. Gontarska, M. Geldenhuys, D. Scheinert, P. Wiesner, A. Polze, and L. Thamsen. “Evaluation of Load Prediction Techniques for Distributed Stream Processing”. In: *IC2E'21*. IEEE. 2021.

[44] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. “GraphX: Graph Processing in a Distributed Dataflow Framework”. In: *OSDI'14*. USENIX, 2014.

[45] A. Gulenko, M. Wallschlager, F. Schmidt, O. Kao, and F. Liu. “Evaluating Machine Learning Algorithms for Anomaly Detection in Clouds”. In: *Big Data'16*. IEEE, 2016.

[46] S. Gupta, C. Fritz, B. Price, R. Hoover, J. Dekleer, and C. Witteveen. "ThroughputScheduler: Learning to Schedule on Heterogeneous Hadoop Clusters". In: *ICAC'13*. USENIX, 2013.

[47] A. Hameed, A. Khoshkbarforoushha, R. Ranjan, P. P. Jayaraman, J. Kolodziej, P. Balaji, S. Zeadally, Q. M. Malluhi, N. Tziritas, A. Vishnu, et al. "A Survey and Taxonomy on Energy Efficient Resource Allocation Techniques for Cloud Computing Systems". In: *Computing* 98.7 (2016).

[48] C. Hardy, E. Le Merrer, and B. Sericola. "Gossiping GANs: Position Paper". In: *DIDL'18*. ACM, 2018.

[49] J. Hasenburg, M. Grambow, E. Grünewald, S. Huk, and D. Bermbach. "Mockfog: Emulating Fog Computing Infrastructure in the Cloud". In: *ICFC'19*. IEEE. 2019.

[50] H. Hawilo, A. Shami, M. Mirahmadi, and R. Asal. "NFV: State of the Art, Challenges, and Implementation in next Generation Mobile Networks (vEPC)". In: *IEEE Netw.* 28.6 (2014).

[51] I. Hegedűs, Á. Berta, L. Kocsis, A. A. Benczúr, and M. Jelasity. "Robust Decentralized Low-Rank Matrix Decomposition". In: *ACM TIST* (2016).

[52] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu. "Serverless Computing: One Step Forward, Two Steps Back". In: *arXiv preprint arXiv:1812.03651* (2018).

[53] T. Hießl, C. Hochreiner, and S. Schulte. "Towards a Framework for Data Stream Processing in the Fog". In: *Informatik Spektrum* 42.4 (2019).

[54] C.-J. Hsu, V. Nair, V. W. Freeh, and T. Menzies. "Arrow: Low-Level Augmented Bayesian Optimization for Finding the Best Cloud VM". In: *ICDCS'18*. IEEE, 2018.

[55] C. Huang, G. Min, Y. Wu, Y. Ying, K. Pei, and Z. Xiang. "Time Series Anomaly Detection for Trustworthy Services in Cloud Computing Systems". In: *IEEE Trans. on Big Data* (2017).

[56] O. Ibidunmoye, A.-R. Rezaie, and E. Elmroth. "Adaptive Anomaly Detection in Performance Metric Streams". In: *IEEE Trans. on Network and Service Management* 15.1 (2018).

[57] G. Janßen, I. Verbitskiy, T. Renner, and L. Thamsen. "Scheduling Stream Processing Tasks on Geo-Distributed Heterogeneous Resources". In: *Big Data'18*. IEEE. 2018.

[58] N. Jazdi. "Cyber Physical Systems in the Context of Industry 4.0". In: *2014 IEEE International Conference on Automation, Quality and Testing, Robotics*. IEEE, 2014.

[59] Y. Jiang, J. Konečný, K. Rush, and S. Kannan. "Improving Federated Learning Personalization via Model Agnostic Meta Learning". In: (2019).

[60] J. Jin, J. Gubbi, S. Marusic, and M. Palaniswami. "An Information Framework for Creating a Smart City through Internet of Things". In: *IEEE Internet Things J.* 1.2 (2014).

[61] A. M. Joy. "Performance Comparison between Linux Containers and Virtual Machines". In: *2015 International Conference on Advances in Computer Engineering and Applications*. IEEE, 2015.

[62] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, Í. Goiri, S. Krishnan, and J. Kulkarni. "Morpheus: Towards Automated Slos for Enterprise Clusters". In: *OSDI'16*. 2016.

[63] P. Kairouz, H. B. McMahan, B. Avent, A. Bellet, M. Bennis, A. N. Bhagoji, K. Bonawitz, Z. Charles, G. Cormode, R. Cummings, et al. "Advances and Open Problems in Federated Learning". In: *arXiv preprint arXiv:1912.04977* (2019).

[64] Y. J. Kim and C. S. Hong. "Blockchain-Based Node-Aware Dynamic Weighting Methods for Improving Federated Learning Performance". In: *APNOMS'19*. 2019.

[65] M. Kleppmann. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. " O'Reilly Media, Inc.", 2017.

[66] J. Konečný, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon. "Federated Learning: Strategies for Improving Communication Efficiency". In: *arXiv preprint arXiv:1610.05492* (2016).

[67] R. S. S. Kumar, A. Wicker, and M. Swann. "Practical Machine Learning for Cloud Intrusion Detection: Challenges and the Way Forward". In: *AISec'17*. ACM, 2017.

[68] A. Lakshman and P. Malik. "Cassandra - a Decentralized Structured Storage System". In: *SIGOPS Oper. Syst. Rev.* 44.2 (2010).

[69] I. Lera, C. Guerrero, and C. Juiz. "Availability-Aware Service Placement Policy in Fog Computing Based on Graph Partitions". In: *IEEE Internet of Things Journal* 6.2 (2018).

[70] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification*. Pearson Education, 2014.

[71] Y. Lu, X. Huang, K. Zhang, S. Maharjan, and Y. Zhang. "Blockchain Empowered Asynchronous Federated Learning for Secure Data Sharing in Internet of Vehicles". In: *IEEE Trans. on Vehicular Technology* (2020).

[72] M. Boehm and M. W. Dusenberry and D. Eriksson and A. V. Evfimievski and F. M. Manshadi and N. Pansare and B. Reinwald and F. R. Reiss and P. Sen and A. C. Surve and S. Tatikonda. "SystemML: Declarative Machine Learning on Spark". In: *Proc. VLDB Endow.* 9.13 (2016).

[73] H. Mao, M. Alizadeh, I. Menache, and S. Kandula. "Resource Management with Deep Reinforcement Learning". In: *HotNets'16*. ACM, 2016.

[74] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh. "Learning Scheduling Algorithms for Data Processing Clusters". In: *SIGCOMM'19*. ACM, 2019.

[75] A. Maros, F. Murai, A. P. Couto da Silva, J. M. Almeida, M. Lattuada, E. Gianniti, M. Hosseini, and D. Ardagna. "Machine Learning for Performance Prediction of Spark Cloud Applications". In: *CLOUD'19*. IEEE, 2019.

[76] E. Masanet, A. Shehabi, N. Lei, S. Smith, and J. Koomey. "Recalibrating global data center energy-use estimates". In: *Science* 367.6481 (2020).

[77] R. Mayer, L. Graser, H. Gupta, E. Saurez, and U. Ramachandran. "Emufog: Extensible and scalable emulation of large-scale fog computing infrastructures". In: *FWC'17*. IEEE. 2017.

[78] A. Mdhaffar, R. B. Halima, M. Jmaiel, and B. Freisleben. "CEP4Cloud: Complex Event Processing for Self-Healing Clouds". In: *WETICE'14*. IEEE, 2014.

[79] A. Mdhaffar, R. B. Halima, M. Jmaiel, and B. Freisleben. "Reactive Performance Monitoring of Cloud Computing Environments". In: *Cluster Comput* 20.3 (2016).

[80] P. M. Mell and T. Grance. *The NIST Definition of Cloud Computing*. National Institute of Standards and Technology, 2011.

[81] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar. "MLlib: Machine Learning in Apache Spark". In: *The Journal of Machine Learning Research* 17.1 (2016).

[82] D. Merkel. "Docker: Lightweight Linux Containers for Consistent Development and Deployment". In: *Linux Journal* 2014.239 (2014).

[83] Milojičić, Dejan and Llorente, Ignacio M. and Montero, Ruben S. "OpenNebula: A Cloud Management Tool". In: *IEEE Internet Comput.* 15.2 (2011).

[84] M. Miyazawa, M. Hayashi, and R. Stadler. "vNMF: Distributed Fault Detection Using Clustering Approach for Network Function Virtualization". In: *IM'15*. IEEE, 2015.

[85] K. Morris. *Infrastructure as Code: Managing Servers in the Cloud.* " O'Reilly Media, Inc.", 2016.

[86] M. Nardelli, V. Cardellini, V. Grassi, and F. L. Presti. "Efficient Operator Placement for Distributed Data Stream Processing Applications". In: *IEEE Trans. on Par. and Dist. Systems* 30.8 (2019).

[87] S. Nastic, T. Rausch, O. Scekic, S. Dustdar, M. Gusev, B. Koteska, M. Kostoska, B. Jakimovski, S. Ristov, and R. Prodan. "A Serverless Real-Time Data Analytics Platform for Edge Computing". In: *IEEE Internet Computing* 21.4 (2017).

[88] S. Nedelkoski, J. Bogatinovski, A. Acker, J. Cardoso, and O. Kao. "Self-Attentive Classification-Based Anomaly Detection in Unstructured Logs". In: *ICDM'20*. IEEE, 2020.

[89] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell. "Samza - Stateful Scalable Stream Processing at LinkedIn". In: *Proc. VLDB Endow.* 10.12 (2017).

[90] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. "Pig Latin: A Not-So-Foreign Language for Data Processing". In: *SIGMOD'08*. ACM, 2008.

[91] R. Ormándi, I. Hegedűs, and M. Jelasity. "Gossip Learning With Linear Models on Fully Distributed Data". In: *Conc. and Comp.: Prac. and Exp.* 25.4 (2013).

[92] A. Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *NeurIPS'19*. Curran Associates, 2019.

[93] T. Pfandzelter and D. Bermbach. "tinyFaaS: A Lightweight FaaS Platform for Edge Environments". In: *2020 IEEE 4th International Conference on Fog and Edge Computing*. IEEE, 2020.

[94] A. Polze and P. Tröger. "Trends and Challenges in Operating Systems — From Parallel Computing to Cloud Computing". In: *Conc. and Comp.: Prac. and Exp.* (2012).

[95] G. J. Popek and R. P. Goldberg. "Formal Requirements for Virtualizable Third Generation Architectures". In: *Commun. ACM* 17.7 (1974).

[96] A. Radovanovic, R. Koningstein, I. Schneider, B. Chen, A. Duarte, B. Roy, D. Xiao, M. Haridasan, P. Hung, N. Care, et al. "Carbon-Aware Computing for Datacenters". In: *arXiv:2106.11750* (2021).

[97] T. Rausch, A. Rashed, and S. Dustdar. "Optimized Container Scheduling for Data-Intensive Serverless Edge Computing". In: *Future Generation Computer Systems* 114 (2021).

[98] F. Rossi, V. Cardellini, and F. L. Presti. "Hierarchical Scaling of Microservices in Kubernetes". In: *ACSOS'20*. IEEE, 2020.

[99] P. J. Sadalage and M. Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Pearson Education, 2013.

[100] M. Satyanarayanan. "The Emergence of Edge Computing". In: *Computer* 50.1 (2017).

[101] D. Scheinert, L. Thamsen, H. Zhu, J. Will, A. Acker, T. Wittkopp, and O. Kao. "Bellamy: Reusing Performance Models for Distributed Dataflow Jobs Across Contexts". In: *CLUSTER'21*. IEEE. 2021.

[102] D. Scheinert, H. Zhu, L. Thamsen, M. K. Geldenhuys, J. Will, A. Acker, and O. Kao. "Enel: Context-Aware Dynamic Scaling of Distributed Dataflow Jobs using Graph Propagation". In: *IPCCC'21*. IEEE. 2021, pages 1–8.

[103] F. Schmidt, F. Suri-Payer, A. Gulenko, M. Wallschlager, A. Acker, and O. Kao. "Unsupervised Anomaly Event Detection for Cloud Monitoring Using Online Arima". In: *UCC'18*. IEEE, 2018.

[104] L. Schuler, S. Jamil, and N. Kühl. "AI-based Resource Allocation: Reinforcement Learning for Adaptive Auto-scaling in Serverless Environments". In: *CCGrid'21*. IEEE, 2021.

[105] O. Sefraoui, M. Aissaoui, and M. Eleuldj. "OpenStack: Toward an Open-Source Solution for Cloud Computing". In: *IJCA* 55.3 (2012).

[106] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. "The Hadoop Distributed File System". In: *MSST'10*. IEEE, 2010.

[107] J. A. Silva, E. R. Faria, R. C. Barros, E. R. Hruschka, A. C. P. L. F. de Carvalho, and J. Gama. "Data Stream Clustering - A Survey". In: *CSUR* 46.1 (2013).

[108] J. Smith and R. Nair. *Virtual Machines*. Elsevier, 2005.

[109] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson. "Container-Based Operating System Virtualization: A Scalable, High-Performance Alternative to Hypervisors". In: *EuroSys'07*. 2007.

[110] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov. "Cloudburst: Stateful Functions-as-a-Service". In: *Proc. VLDB Endow.* 13.12 (2020).

[111] M. Symeonides, Z. Georgiou, D. Trihinas, G. Pallis, and M. D. Dikaiakos. "Fogify: A fog computing emulation framework". In: *SEC'20*. IEEE. 2020.

[112] A. S. Tanenbaum and H. Bos. *Modern Operating Systems*. 4th. Prentice Hall Press, 2014.

[113] L. Thamsen, J. Beilharz, V. T. Tran, S. Nedelkoski, and O. Kao. "Mary, Hugo, and Hugo*: Learning to Schedule Distributed Data-Parallel Processing Jobs on Shared Clusters". In: *Conc. and Comp.: Prac. and Exp.* 33.18 (2020), e5823.

[114] L. Thamsen, B. Rabier, F. Schmidt, T. Renner, and O. Kao. "Scheduling Recurring Distributed Dataflow Jobs Based on Resource Utilization and Interference". In: *BigData Congress'17*. IEEE, 2017.

[115] L. Thamsen, T. Renner, I. Verbitskiy, and O. Kao. "Adaptive Resource Management for Distributed Data Analytics". In: *Big Data and HPC: Ecosystem and Convergence* 33 (2018).

[116] L. Thamsen, I. Verbitskiy, F. Schmidt, T. Renner, and O. Kao. "Selecting Resources for Distributed Dataflow Systems According to Runtime Targets". In: *IPCCC'16*. IEEE, 2016.

[117] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. "Hive: A Warehousing Solution over a Map-Reduce Framework". In: *Proc. VLDB Endow.* 2.2 (2009).

[118] L. Toka, G. Dobreff, B. Fodor, and B. Sonkoly. "Adaptive AI-based autoscaling for Kubernetes". In: *CCGRID'20*. IEEE, 2020.

[119] A. Toshniwal, J. Donham, N. Bhagat, S. Mittal, D. Ryaboy, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, and M. Fu. "Storm@twitter". In: *SIGMOD'14*. ACM, 2014.

[120] R. Uhlig, G. Neiger, D. Rodgers, A. Santoni, F. Martins, A. Anderson, S. Bennett, A. Kagi, F. Leung, and L. Smith. "Intel Virtualization Technology". In: *Computer* 38.5 (2005).

[121] P. Vanhaesebrouck, A. Bellet, and M. Tommasi. "Decentralized Collaborative Learning of Personalized Models over Networks". In: *AISTATS'17*. 2017.

[122] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica. "Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics". In: *NSDI'16*. 2016.

[123] A. Verma, L. Cherkasova, and R. H. Campbell. "Aria - Automatic Resource Inference and Allocation for Mapreduce Environments". In: *ICAC'11*. ACM, 2011.

[124] Z. Wang, M. Song, Z. Zhang, Y. Song, Q. Wang, and H. Qi. "Beyond Inferring Class Representatives: User-Level Privacy Leakage from Federated Learning". In: *CoRR* (2018).

[125] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. "Ceph: A Scalable, High-Performance Distributed File System". In: *OSDI'06*. USENIX, 2006.

[126] J. Weng, J. Weng, J. Zhang, M. Li, Y. Zhang, and W. Luo. "DeepChain: Auditable and Privacy-Preserving Deep Learning with Blockchain-Based Incentive". In: *IEEE Trans. on Dependable and Secure Computing* (2019).

[127] P. Wiesner, I. Behnke, D. Scheinert, K. Gontarska, and L. Thamsen. "Let's Wait Awhile: How Temporal Workload Shifting Can Reduce Carbon Emissions in the Cloud". In: *Middleware'21*. ACM, 2021.

[128] P. Wiesner and L. Thamsen. "LEAF: Simulating Large Energy-Aware Fog Computing Environments". In: *ICFEC'21*. IEEE. 2021.

[129] J. Will, L. Thamsen, D. Scheinert, J. Bader, and O. Kao. "C3O: Collaborative Cluster Configuration Optimization for Distributed Data Processing in Public Clouds". In: *IC2E'21*. IEEE. 2021.

[130] C. Wu, J. M. Faleiro, Y. Lin, and J. M. Hellerstein. "Anna: A KVS for Any Scale". In: *IEEE Trans. on Knowledge and Data Engineering* 33.2 (2021).

[131] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. "Shark: SQL and Rich Analytics at Scale". In: *SIGMOD'13*. ACM, 2013.

[132] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. "Spark: Cluster Computing with Working Sets". In: *HotCloud'10*. USENIX, 2010.

[133] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. "Discretized Streams - Fault-Tolerant Streaming Computation at Scale". In: *SOSP'13*. ACM, 2013.

[134] M. Zekri, S. E. Kafhali, N. Aboutabit, and Y. Saadi. "DDoS Attack Detection Using Machine Learning Techniques in Cloud Computing Environments". In: *CloudTech'17*. IEEE, 2017.

[135] S. Zeuch, A. Chaudhary, B. Del Monte, H. Gavriilidis, D. Giouroukis, P. M. Grulich, S. Breß, J. Traub, and V. Markl. “The NebulaStream Platform: Data and Application Management for the Internet of Things”. In: *CIDR'19*. CIDR, 2019.

[136] S. Zhang, T. Wu, M. Pan, C. Zhang, and Y. Yu. “A-SARSA: A Predictive Container Auto-Scaling Algorithm Based on Reinforcement Learning”. In: *ICWS'20*. IEEE, 2020.

[137] J. Zheng, A. A. Chien, and S. Suh. “Mitigating Curtailment and Carbon Emissions through Load Migration between Data Centers”. In: *Joule* 4.10 (2020).