Continuous Big Data Applications in Industry[*]

– Modern Development Tools, Distributed Operational and Orchestration Systems, Internet of Things –

Prof. Dr. Victor David Sánchez, Ph.D.
Brilliant Brains. Palo Alto, California

December, 2016

Abstract

The abundance and continuous growth of data volumes offer concurrently business opportunities as well as computational challenges. To meet the latter, solutions to associated computations needed to be moved from single processors to multiple node clusters with often specialized programming models. The author has been very actively and concretely involved in that moving process, indirectly pushing the Intels and Crays of the world to follow, what they finally did in their own interest only after decades, that process is still in its infancy. For example, the Map-Reduce model is used in Hadoop and with optimized performance in Spark for data loading. Big data and the Internet of Things (IoT) data processing require not only a model for batch mode, but also for continuous applications, SQL-like queries, and iterative machine learning inference and training. Since processing in batch mode is easier, we focus on the development of continuous applications, which require comfortable Integrated Development Environments (IDEs), consisting of different components that play well together. This leads to the offerings of so called stacks, e.g., the Spark stack. Spark can be run as standalone cluster, on Amazon EC2, on Hadoop YARN, on Apache Mesos; can access data in HDFS, Cassandra, HBase, Hive, S3; specialized libraries implemented over the core engine include Spark SQL, Spark Streaming, ML, Graph. The associated ecosystems are extremely dynamic, i.e., individual components are constantly being upgraded in functionality and their interoperability needs to be constantly verified. Later we provide further examples of software service stacks and vertical industrial applications.

After building ASIC-based custom multiprocessors and Real-Time Operating Systems (RTOS) for automation at Siemens AG, the author designed and built at the German NASA (DLR) an scalable, heterogeneous, parallel distributed supercomputer given special requirements imposed by space missions to provide real-time Artificial Intelligence (AI) for space and terrestrial applications. Figure 1 shows in (a) a space Augmented/Virtual Reality (AR/VR) application flown with NASA's spaceshuttle and ESA's spacelab aboard, in (b) an early prototype of a real-time supercomputer I designed and built, in its final version we used it for the corresponding NASA-ESA-DLR mission flown with the Spaceshuttle Columbia, and in (c) an outline of the workload partitioning between the U.S.A. and Europe for that space mission. This real-time parallel distributed supercomputer, the fastest worldwide of its time, which I originally wanted to build at Siemens AG Corporate R&D in Munich after I had already been building custom processors for automation based among others on ASICs which we designed with an AI-based tool for VLSI I developed at the Karlsruhe Institute of Technology (KIT), I finally built with industrial participation at the German NASA (DLR) where I was civil servant for about a decade. That real-time, scalable, heterogeneous, parallel distributed supercomputer architecture was well ahead of its time worldwide, several decades ahead as we all know now, and was only possible with the participation of national labs and industry under my hands-on leadership. Most requirements currently placed for continuous big data applications in industry were fulfilled and operational with that architecture and the lessons learned still far ahead of the best environments today, whose functionalities in hardware and software need to be substantially improved to offer hard real-time computer vision and machine learning for specific tasks just to mention an example from the wish list.

REFERENCES

[1] Amazon. Deploying Java Microservices on Amazon Elastic Container Service. https://aws.amazon.com/blogs/compute/deploying-java-microservices-on-amazon-ec2-container-service/.

[*]This abstract has been granted permission for public release.

[2] Databricks. Mastering Apache Spark 2.0, 2016.

[3] Docker. Docker Documentation. `https://docs.docker.com/`.

[4] B. Hindman et al. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center, UCB/EECS-2010-87, 2010.

[5] Forrester, commissioned by Microsoft. DevOps Best Practices – The Path to Better Application Delivery Results –, 2015.

[6] Mesosphere. ESRI builds flexible Real-Time Mapping Managed Service integrating Cloud, Container, IoT, and GIS Technologies, 2016.

[7] V.D. Sánchez. Neurocomputers in Industrial Applications (in German); Neurocomputing–Research and Applications–; Increasing the Autonomy of Space Robots; Intelligent BioSystems; Modeling Dynamics for Communications, Navigation, Guidance and Control Applications. Technische Rundschau 82[65], German Research Center for Anthropotechnics, Wachtenberg-Werthoven, Germany; NASA Ames Research Center, Moffett Field, CA; KAIST Department of BioSystems, Daejeon, South Korea; Rockwell Collins, Advanced Technology Center (ATC), Cedar Rapids, IA, 1990, 1990, 2002, 2003, 2011.

[8] V.D. Sánchez. A Parallel Distributed Architecture for Vision and Telerobotics (in German), TAT'92, Springer-Verlag, 1993.

[9] V.D. Sánchez. The Interplanetary Internet of Things – Use Case: Descent in Celestial Bodies –, Brilliant Brains, TR, 2016.

[10] Chef Software. DevOps and the Cloud – Chef and Microsoft Azure –, 2015.

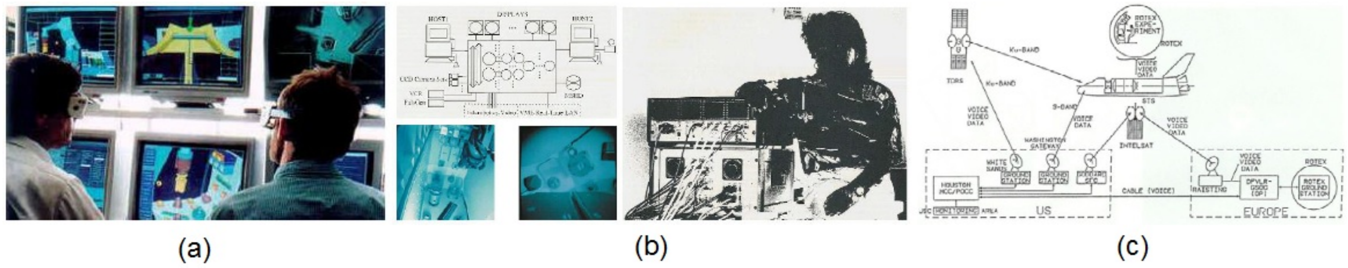


Figure 1: Scalable, Heterogeneous, Parallel Distributed Architecture to process continuous spaceborne data in real-time during mission flown with NASA's Spaceshuttle and ESA's Spacelab aboard [8]

DevOps For this category of applications, we first summarize basic concepts, technologies, services, and platforms associated to DevOps, containerization, microservices, and cloud continuous computing, and then describe concrete examples of the integration of components in the current development of vertical applications in industry. DevOps' overall goal is to unify software development and operations, and is often viewed as an approach to efficient system development and administration of software in the cloud. This includes all modern variants, like for example DevSecOps, where security is part of quality, and quality assurance is part of software development in this abbreviated form of concept understanding. For DevOps to be effective, Architecturally Significant Requirements (ASR) of software applications need to be met, which include among others deployability, modifiability, testability, and monitorability. The following recommendations for DevOps practices resulted from in-depth surveys with application development and IT operation professionals [5]: (1) Streamline, simplify, and automate the delivery pipeline, (2) Expand test automation to improve quality while increasing delivery speed, (3) Use infrastructure as code and cloud technologies to simplify and streamline environment provisioning, (4) Reduce technical debt to increase responsiveness and reduce cost, (5) Decouple applications and architectures to simplify delivery activities, (6) Collect and analyze feedback to drive better requirements, and (7) Measure business outcomes tied to application releases.

DevOps includes, involves several activities, areas including continuous integration, continuous delivery (CI/CD), cloud infrastructure, test automation, and configuration management. Those efforts lead once mastered to, e.g., more than 10 releases/week at Facebook, Netflix, just to mention one example, the latter completing integration, testing, and delivery in 2-3 hrs. Figure 2 shows in (a) DevOps solutions (right side) to pervasive problems (left side) with traditional IT practices, and in (b) the role and value of automation to support best DevOps patterns and practices [10]. A primary technology that enables a fluent DevOps workflow is automation, which allows the description of the infrastructure as code. One can then eliminate error-prone, time-consuming manual tasks, standardize development, test, and production environments, build automated release pipelines, and improve cooperation between development and operations. Code is versionable, testable, and repeatable. Then the deployment pipeline for the infrastructure and the applications should be the same. Once the infrastructure is also code, all advantages that apply to code can be utilized for the infrastructure as well, among many others, e.g., transparency enhancement by using source control to examine differences between configuration versions and show what precisely has changed since the previous stable system version.

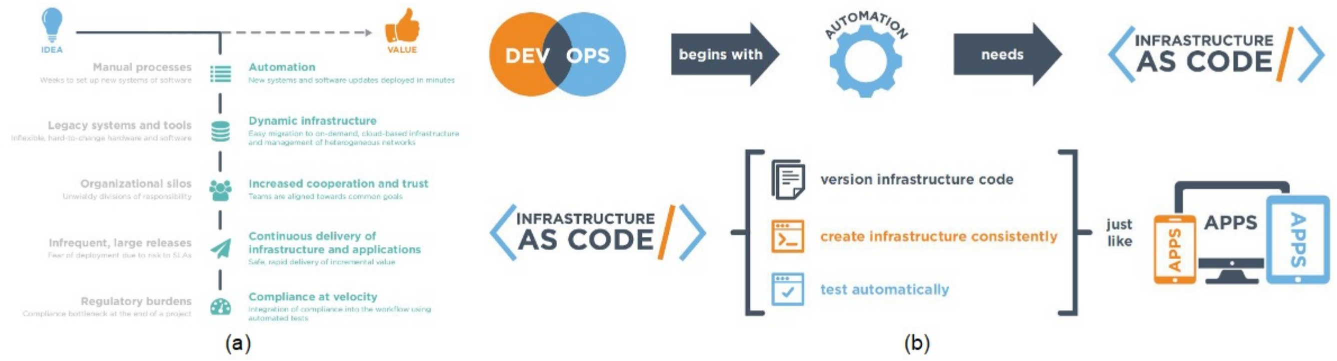


Figure 2: DevOps solutions and the key role and value of Automation in DevOps [10]

Containers Microservices are currently a preferred architecture to build continuously deployed software systems due to the small service size which allows for fast and continuous refactoring, release, and deployment as well as align well with recommended DevOps best practices after traditional IT practices were not designed for the flexibility and speed that the cloud offers. On the other hand, in particular, the concept of containerized application is very popular these days, in DevOps too. Docker [3] is for example a software technology that allows containerization of applications, i.e., allows packaging an application together with all its dependencies so that it can run on any infrastructure with almost no configuration changes on the customer premises or in the cloud. It does so by implementing a high-level API to provide lightweight containers that run processes in isolation. Figure 3 shows in (a) the architecture difference between a Virtual Machine (VM) on the left and a Docker container on the right, in (b) the components of a Docker engine, a client-server application with a long running daemon process as a server, a REST API for programs to issue instructions to the daemon, and a Command Line Interface (CLI) client, and in (c) the different interfaces that Docker uses to access virtualization features of the Linux kernel. Docker can be integrated with a variety of infrastructure tools including but not limited to Amazon Web Services (AWS), Google Compute Engine (GCE), Microsoft Azure, Kubernetes, to mention just a few. Next, we connect containers to services and microservices.

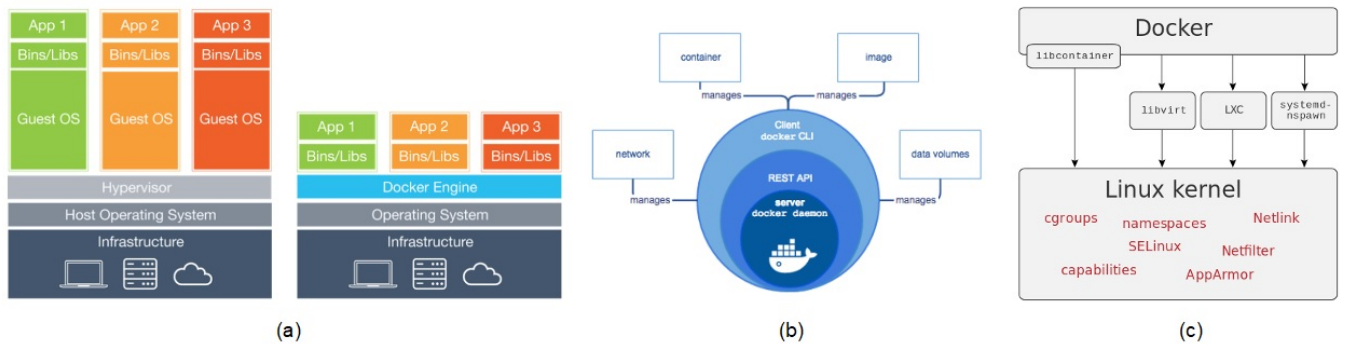


Figure 3: Docker architecture, Docker engine components, and Docker interfaces to access virtualization [3]

Services As an example, Amazon Web Services (AWS), a subsidiary of Amazon, offers on-demand cloud computing platforms, i.e., full-fledged virtual and real clusters of computers available all the time through the Internet, which include among many others access to CPUs and GPUs as well as a choice of operating systems, pre-loaded application software, and API-accessible services in those clusters. The main access to those resources is provided via a browser login over the Hypertext Transfer Protocol (HTTP) using Representational State Transfer (REST) and the Simple Object Access Protocol (SOAP). AWS subscribers can then choose to deploy the subscribed AWS systems to provide the subscribers' own Internet-based services for their own customers. The most popular AWS services are the Elastic Compute Cloud (EC2) and the Simple Storage Service (S3), offered since 2006. The AWS subscribers can develop their own applications, services and deploy scalable applications using a web service through which an Amazon Machine Image (AMI) to configure a virtual machine, i.e., an instance, containing any software desired, can be booted. Additional examples of services relevant for the explanatory context here are the Amazon Virtual Private Cloud (VPC), which creates a logically isolated set of AWS resources that can be connected using a Virtual Private Network (VPN) connection, the Amazon Relational Database Service (RDS), which provides scalable database servers with support for MySQL, Oracle among others, and the Amazon EC2 Container Service (ECS), a highly scalable and fast container management service using Docker containers.

MICROSERVICES The following example shows how to containerize a monolithic Java application to run on Docker, deploy it on AWS using ECS, and how to convert the monolith into multiple microservices, all running in containers on ECS. Figure 4 shows in (a) a simple 3-tier architecture for an application example: a client (curl simulation), a web server (Java/Spring/Tomcat), and a database server (MySQL RDS), in (b) an overview of the setup for ECS, the architecture is containerized but still monolithic because each container has all the same features as the rest of the containers, and in (c) a container deployment overview after converting the monolithic application into microservices running on Amazon ECS [1].

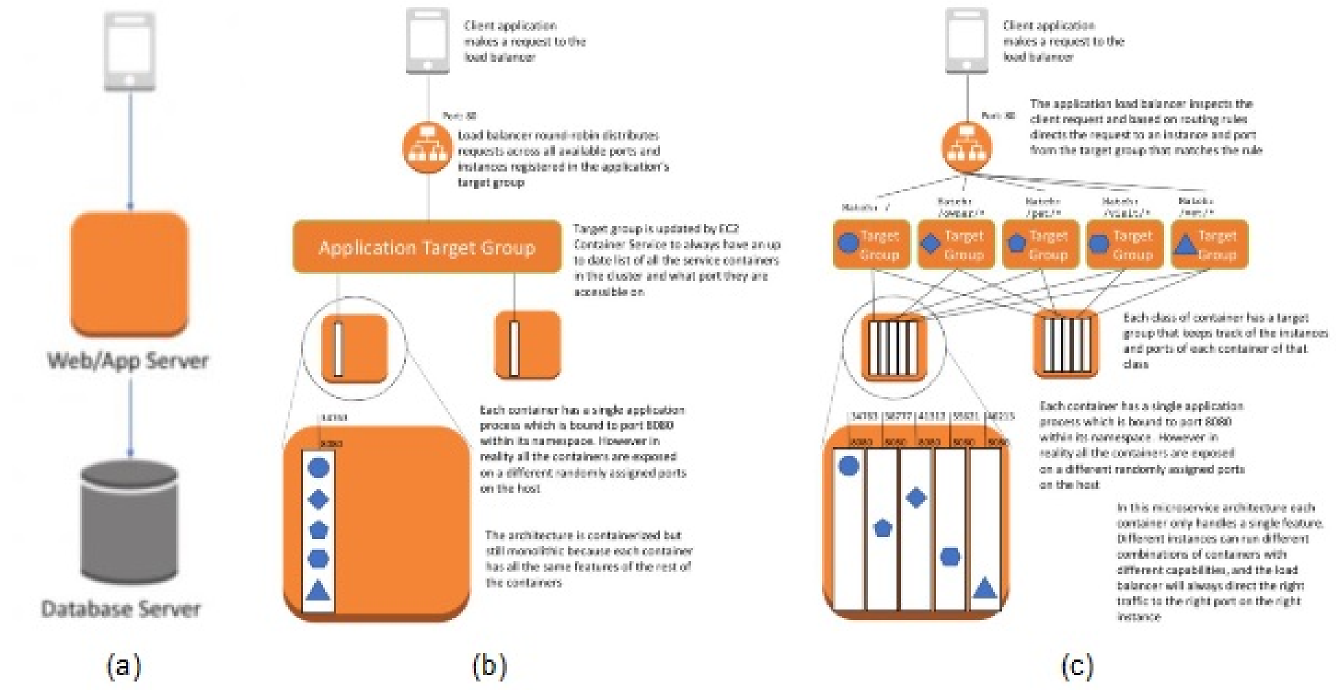

Web/App Server
Database Server
(a)
Client application makes a request to the load balancer
Port 80
Load balancer round-robin distributes requests across all available ports and instances registered in the application's target group
Application Target Group
Target group is updated by EC2 Container Service to always have an up to date list of all the service containers in the cluster and what port they are accessible on
32768
8080
Each container has a single application process which is bound to port 8080 within its namespace. However in reality all the containers are exposed on a different randomly assigned ports on the host
The architecture is containerized but still monolithic because each container has all the same features of the rest of the containers
(b)
Client application makes a request to the load balancer
Port 80
The application load balancer inspects the client request and based on routing rules directs the request to an instance and port from the target group that matches the rule
Matches /
Matches /owner/*
Matches /pet/*
Matches /visits/*
Matches /vet/*
Target Group
Target Group
Target Group
Target Group
Target Group
Each class of container has a target group that keeps track of the instances and ports of each container of that class
34783
58717
41312
55631
48215
8080
8080
8080
8080
8080
Each container has a single application process which is bound to port 8080 within its namespace. However in reality all the containers are exposed on a different randomly assigned ports on the host
In this microservice architecture each container only handles a single feature. Different instances can run different combinations of containers with different capabilities, and the load balancer will always direct the right traffic to the right port on the right instance
(c)

Figure 4: Microservices, Docker-containerized, deployed on Amazon Elastic Container Service (ECS) [1]

In the case of the monolithic application, given that an ECS cluster hosts the container for the application and a VPC network hosts the ECS cluster and associated security groups, the monolithic application runs on ECS as follows, cf. Figure 4(b). A client application makes a request to the load balancer. The load balancer round-robin distributes requests across all available ports and instances registered in the application's target group. The application target group is updated by ECS to always have an up to date list of all service containers in the cluster and what port they are accessible on. Each container has a single application process which in this example is bound to port 8080 within its namespace. In reality, all containers are exposed on different randomly assigned ports on the host. The architecture is containerized but still monolithic because each container has all the same features as the rest of the containers. For the application to run, one needs in addition: one Amazon Elastic Container Registry (ECR) repository for the application, a service/task definition that spins up containers on the instances of the ECS cluster, and a MySQL RDS instance that hosts the application's schema. The information about the MySQL RDS instance is sent in through environment variables to the containers, so that the application can connect to the MySQL RDS instance.

In the case of the microservices application, given that an ECS cluster hosts the container for each microservice of the application, a VPC network hosts the ECS cluster and associated security groups, and once converted, the microservices application runs on ECS as follows, cf. Figure 4(c). A client application makes a request to the load balancer. The load balancer inspects the request and based on routing rules directs the request to an instance and port from the target group that matches the rule. Each class of container has a target group that keeps track of the instances and ports of each container of that class. Each container has a single application process which in this example is bound to port 8080 within its namespace. In reality, all containers are exposed on different randomly assigned ports on the host. In this microservice architecture each container only handles a single feature. Different instances can run different combinations of containers with different capabilities. The load balancer directs the right traffic to the right port on the right instance. For the application to run, one needs in addition: one ECR repository for each microservice, a service/task definition per microservice that spins up containers on the instances of the ECS cluster, and a MySQL RDS instance that hosts the application's schema.

CONTINUOUS APPLICATIONS The use of specific tools to develop real-time, streaming and continuous applications can substantially facilitate the endeavor. In this context, a continuous application can be a streaming application, but can also be a larger application that includes streaming, serving, storage, and batch jobs. A continuous application can update data served in real time (not a stream map-reduce in this case), can extract, transform and load (ETL), e.g., from JSON logs to Hive table, can create a real-time version of an existing batch job, and can perform online machine learning, i.e., combining large static datasets processed using batch jobs with real-time data and live prediction serving. For an example of such an above-referenced tool, new features in Spark 2.0 support the overall purpose of simplifying development, which are included within the Structured Streaming API for building continuous applications and which are integrated into Spark's Dataset and DataFrame APIs. Issues of consistency, fault tolerance, and out-of-order data have been taken into account and solved. At any time, the application's output is the same as executing an equivalent batch job on a prefix of the data.

Existing methods to transform data can be used including map, filter, select, run aggregations like count, windowed aggregations, joining streams with static data, and interactive queries. Figure 5 shows in (a) Spark's structured processing model: users express queries using a batch API while Spark incrementalizes them to run on streams, in (b) conceptual treatment of all data arriving by Spark's structured streaming as an unbounded input table, where each new stream item is like a row appended to the input table, and in (c) an example of a map-reduce pattern node setup for data processing in a distributed streaming engine where mappers and reducers are respectively split by and count by (action, hour). With the Structured Streaming API, we can analyze IoT streaming data emanating for example from 4 device types: sensor-igauge, sensor-inest, sensor-ipad, and sensor-istick. Real-time metrics on a stream of timestamped device events, e.g., running and windowed counts, can be computed using the Structured Streaming API as shown in Figure 6, in (a) the unique device type count, and in (b) a continuous aggregation, i.e., running averages of a particular device signal strength.

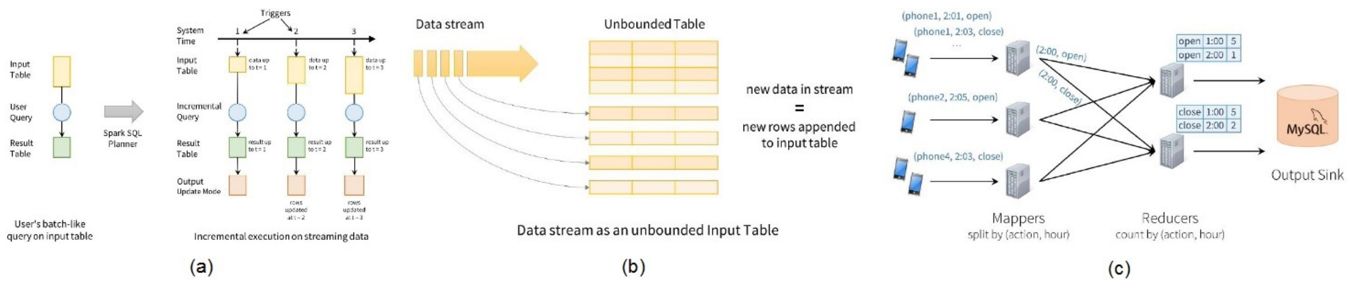


Figure 5: Distributed processing of a continuous application [2]

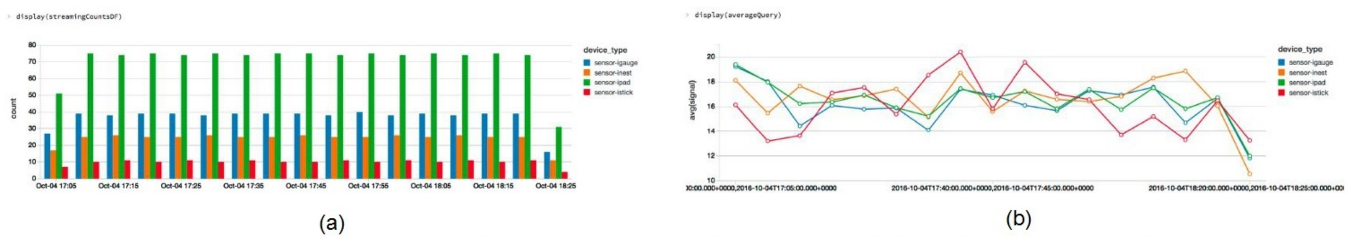


Figure 6: Determining real-time metrics from IoT device event data [2]

DATACENTER OPERATING SYSTEM The Mesosphere's Datacenter Operating System (DC/OS) is based on the general purpose cluster manager Apache Mesos [4]. DC/OS is itself a distributed system, a cluster manager, a container platform, and an operating system. DC/OS abstracts the cluster hardware and software resources to provide common services to applications, which include cluster management, container orchestration, package management, networking, logging, and metrics, storage and volumes, and identity management. DC/OS has a kernel space, which is a protected area, used for resource allocation, security, process isolation, and a user space. which is the area where the user applications, jobs, and services run. DC/OS spans multiple machines and relies on each machine having its own host OS and host kernel. Services can be installed into the user space using DC/OS' built-in package manager.

Among others, DC/OS performs container orchestration and can elastically run microservices in production, including both containers and stateful data services. Figure 7 shows in (a) the main Mesos components: a master process managing slave daemons running on each cluster node and frameworks running tasks on the slaves, here two frameworks are shown: Hadoop and MPI, in (b) how a framework is scheduled to run tasks after the slave reports its resources to the master, the master invokes the allocation module and sends a resource offer to framework 1 in the example, the framework's scheduler replies with information about two tasks to run on the slave, and the master sends the two tasks to the slave, and in (c) the DC/OS ecosystem with DC/OS as the middle component of three, the top component includes different data services: Spark, Hadoop, Elasticsearch and others as shown, and the bottom component which represents hybrid cloud portability including the public cloud providers.

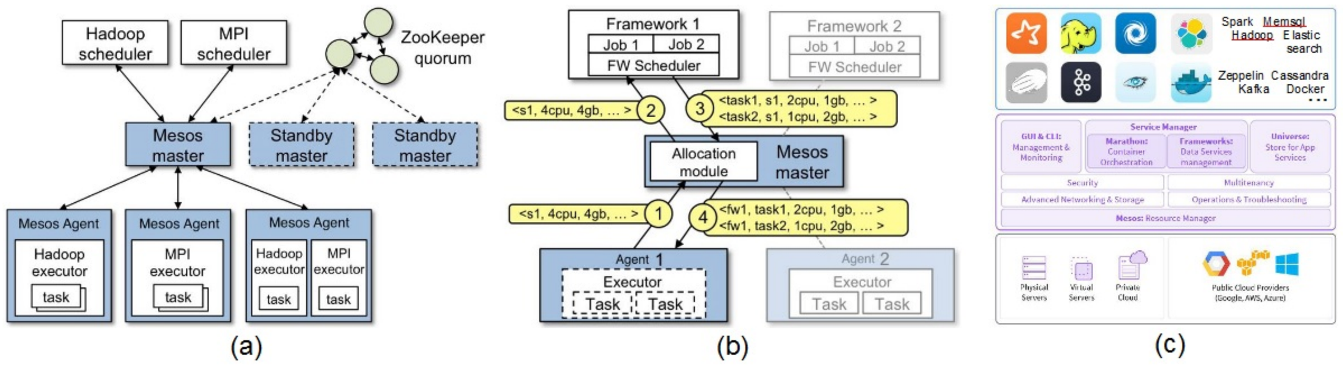


Figure 7: Cluster and resource management in a datacenter using Mesos [4]

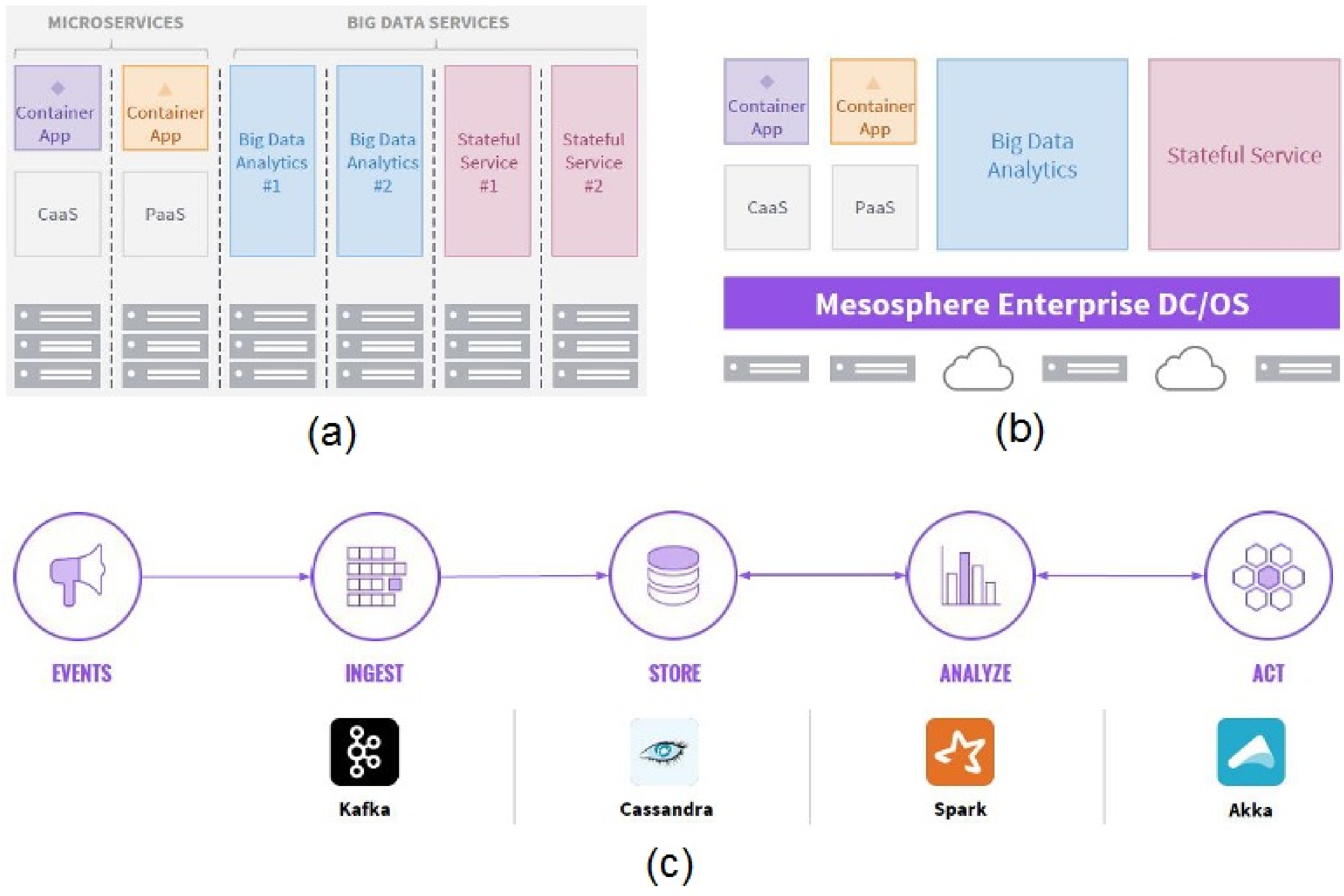


Figure 8: Service orchestration of a continuous application using DC/OS

Figure 8 shows in (a) the multiplexing of data, services, user, and environments in a datacenter without Mesos and DC/OS: siloed, over-provisioned servers, low utilization, in (b) such multiplexing using Mesos and DC/OS: automated schedulers, workload multiplexing onto the same machines, and in (c) how streaming event data processing can be performed using Kafka for the data ingest, Cassandra for data storage, Spark for data analysis, and Akka to act as data sink. For that type of scenario, the SMACK stack used is the convergence of complementary technologies: Spark, Mesos, Akka, Cassandra, and Kafka. Another example involves using Google's TensorFlow to train deep learning models and Spark on a cluster of machines under Mesos coordination to improve deep learning pipelines with TensorFlow during model hyperparameter tuning and deploying models at scale. Artificial Intelligence (AI) and Machine Learning (ML) are enjoying a tremendous boost being introduced as integral solutions to problems in such diverse industries as biotechnology, finance, multimedia, cloud, and cybersecurity among others. The author has contributed for many years to the introduction of these technologies to pioneering industrial applications [7], more recently extending the most sophisticated and latent applications, e.g., in Big Data and IIoT (Industrial Internet of Things) and related, such as Industrie 4.0 named in reference to an on-going 4th industrial revolution [9].

Figure 9 shows in (a) a digit recognition solved by deep learning modeling and its model hyperparameter tuning to select the best model, in this example a 34% reduction of the test error was accomplished, in (b) the computation speedup using a 13-node cluster, achieved was a 7x speedup in this example, and in (c) the sensibility analysis of model hyperparameters shows that the neuron number is not so important in this example and the criticality of the learning rate demands that the value chosen neither be too low nor too high.

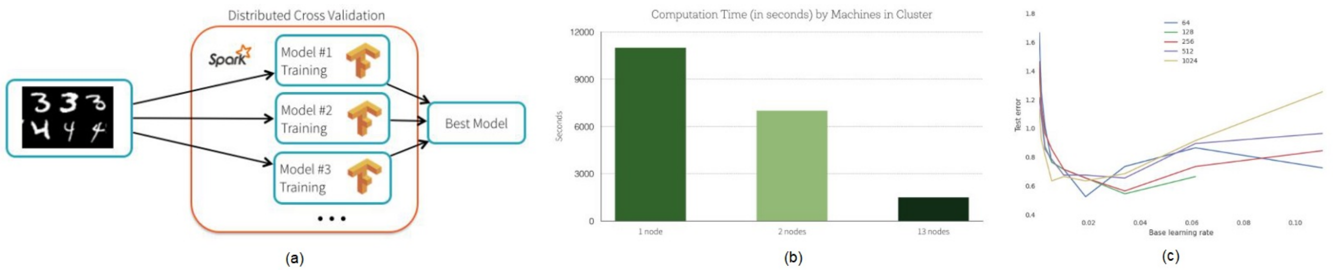
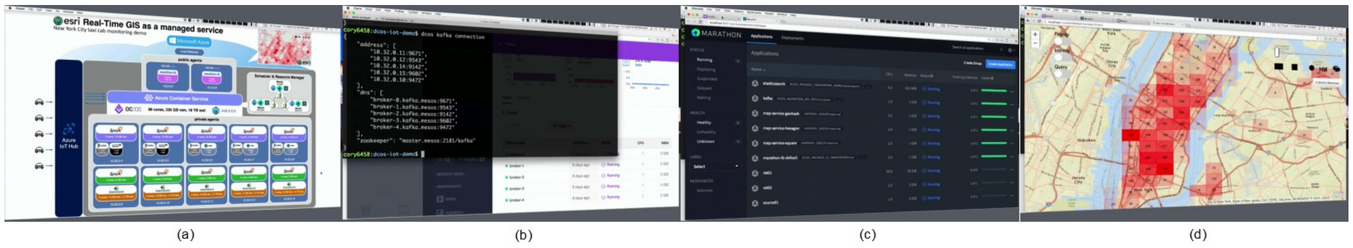


Figure 9: Deep Learning using TensorFlow, Spark, Mesos, GPUs

Vertical Applications To illustrate the integration of services when developing a vertical application, a Geographic Information System (GIS) application is outlined. ESRI, a GIS market leader, solved its challenge to map Internet of Things (IoT) data streaming in real-time by combining cloud, container, IoT, and GIS technologies [6]. Its customer base had the desire to connect millions of sensors to a platform where they can visualize the data on a map in real time. To fulfill scale and performance requirements, Microsoft's Azure Cloud, Container Service, and IoT Suite on top of Mesosphere's DC/OS was chosen to be able to focus more on own software component development and less on the infrastructure it runs on. Its internal software required also flexibility to support multiple container technologies including Docker and Mesos. The solution uses several frameworks including Kafka, Spark Streaming, Elasticsearch, Marathon, Chronos, and Lightbend Reactive Platform. It enables users to perform real-time, predictive mapping at any scale required.

Figure 10 shows in (a) the architecture of the ESRI's real-time big data mapping managed service integrating cloud, container, IoT, and GIS technologies, in (b) in the foreground the Command Line Interface (CLI) of DC/OS used to configure the Kafka service showing the 5 brokers deployed and in the background the Graphical User Interface (GUI) display of the Kafka package when opened from the graphical overview of packages installed by DC/OS for a demonstration, in (c) the GUI display of the Marathon package when opened from the graphical overview of packages installed by DC/OS for the same demonstration, and in (d) the real-time mapping application showing the aggregation of data on the fly after performing analytics by Spark Streaming from raw data streaming taken by Kafka and sending results to Elasticsearch.



```
cory6458:dcos-iot-demo$ dcos kafka connection
{
  "address": [
    "10.32.0.11:9671",
    "10.32.0.12:9543",
    "10.32.0.14:9142",
    "10.32.0.15:9602",
    "10.32.0.10:9472"
  ],
  "dns": [
    "broker-0.kafka.mesos:9671",
    "broker-1.kafka.mesos:9543",
    "broker-2.kafka.mesos:9142",
    "broker-3.kafka.mesos:9602",
    "broker-4.kafka.mesos:9472"
  ],
  "zookeeper": "master.mesos:2181/kafka"
}
cory6458:dcos-iot-demo$
```

Figure 10: Big Data Mapping Managed Service integrating Cloud, Container, IoT, and GIS Technologies [6]

Common sense shows that applications that can be delivered on a real-time, scalable, heterogeneous, parallel distributed architecture like the one referred to before [8], with diverse built-in processors and programming models as we realized them continues to pave the way for further, more challenging continuous Big Data applications in industry running on CPUs, DSPs, GPUs, FPGAs, and neurochips, where for example, video-streaming does not necessarily have to be coming down from space.