ICT-32-2014: Cybersecurity, Trustworthy ICT

WITDOM
"empoWering prIvacy and securiTy in non-trusteD envirOnMents"

D4.6 – Final specification of the adaptation layer for Cloud computing

Due date of deliverable: 31 October 2017

Actual submission date: 31 October 2017

Grant agreement number: 644371
Start date of project: 1 January 2015
Revision 1.0

Lead contractor: Atos Spain SA (ATOS)
Duration: 36 months

Project co-funded by the European Commission within the EU Framework Programme for Research and Innovation HORIZON 2020, and the Swiss State Secretariat for Education, Research and Innovation (SERI)	
Dissemination Level	
PU = Public, fully open	X
CO = Confidential, restricted under conditions set out in the Grant Agreement	
CI = Classified, information as referred to in Commission Decision 2001/844/EC	

D4.6

Final specification of the adaptation layer for Cloud computing

Editors

Marcus Brandenburger (IBM)

Contributors

Juan Troncoso (UVIGO), Lilian Adkinson (UVIGO), Pablo Dago (UVIGO),

Gonzalo Jiménez (UVIGO), Bruno Fernández (UVIGO),

Eduardo Gonzalez Real (ATOS), Nicolas Notario (ATOS),

Jolanda Modic (XLAB), Manca Bizjak (XLAB), Miha Stopar (XLAB),

Marcus Brandenburger (IBM), Christian Cachin (IBM), Eduarda Freire (IBM), Mike Osborne (IBM)

Reviewers

Elsa Prieto Perez (ATOS), Eleonora Ciceri (FCSR)

31 October 2017
Revision 1.0

The work described in this document has been conducted within the project WITDOM, started in January 2015. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 64437. This work was supported in part by the Swiss State Secretariat for Education, Research and Innovation (SERI) under contract number 15.0098.

The opinions expressed and arguments employed herein do not necessarily reflect the official views of the European Commission or the Swiss Government.

Executive Summary

The architectural work package (WP4) of WITDOM develops a flexible end-to-end secure architecture that forms the basis for securing use-case applications, and can also be adapted to a cloud computing framework. This deliverable provides the final specification of an adaptation layer for the generic architecture proposed in Deliverable D4.2 to cloud computing environments. This means that here the WITDOM architecture is specified for the case when untrusted environments, to which data storage and processing are outsourced, is the cloud. With this intent, different cloud services and deployment models are investigated considering the requirements of the WITDOM scenarios and architectural challenges for the selected models are addressed.

The adaptation layer for cloud computing presented in this deliverable considers a hybrid cloud model used for WITDOM, where the trusted domain relies on a private cloud infrastructures, such as OpenStack, while the untrusted domain benefits from the use of public clouds, such as Amazon Web Services. A private cloud enables end-users of the WITDOM architecture to easily automate and scale their resources according to their requirements, while at the same time keeping certain processing in a safe (trusted) environment. On the other hand, the use of a public cloud in the untrusted domain gives organizations the advantage of instantly scaling their infrastructure up or down, avoiding capital investment on on-premises infrastructure, and also reducing costs of maintenance for IT resources.

All components in the hybrid cloud are provisioned and managed by an open-source cloud orchestrator software, named *Cloudify*, which allows WITDOM users to automate the deployment of WITDOM components and efficient usage of the infrastructure. Components are packaged and deployed in *software containers*, using for example the open-source platform *Docker*, to enable easy deployment of dependencies and continuous integration. The adapted architecture to the cloud also enables protection components to benefit from big data processing mechanisms, e.g., distributed and parallel processing of large volumes of input data, which is extremely important in the e-Health and financial scenarios. For this, a *big data component* is added as an extra component in the WITDOM architecture.

This deliverable specifies in detail how the core components and protection components are adapted to the cloud computing framework. Some components, such as identity and access management (IAM) and key management (KM), can be instantiated by well-established open-source components, like OpenStack *Keystone* and OpenStack *Barbican*. The deployment of other components developed within WITDOM, such as the broker, protection orchestrator, and protection components need to be slightly adapted.

Moreover, this deliverable also shows how cloud-based computing improves operations related to individual e-Health (sequence alignment and variant annotation) and financial scenarios (fraud detection and risk scoring), and provides implementation guidelines corresponding to the cloud scenario.

Finally, architectural challenges for the selected models are addressed: from remote management and administration of the overall solution to specific technical issues stemming from the use of cloud.

Contents

List of Figures

List of Tables

Glossary

API	Application Program Interface
AWS	Amazon Web Services
Broker	Agent that mediates services and requests among components
CI	Continuous Integration
DOA	Description of Action
E2EE	End-to-end Encryption
FASTQ	Text-based file format for storing nucleotide information and quality
FHE	Fully Homomorphic Encryption
FPGA	Field-Programmable Gate Array, an integrated circuit that can be programmed at runtime
GUI	Graphical User Interface
HE	Homomorphic Encryption
HSM	Hardware Security Module
IAM	Identity and Access Manager
ICT	Information and Communication Technology
IT	Information Technology
IdP	Identity Provider
JCA	Java Cryptography Architecture
JNI	Java Native Interface
JVM	Java Virtual Machine
KM	Key Management
KMIP	Key Management Interoperability Protocol
LDAP	Lightweight Direct Access Protocol
OAuth	Open standard for authorization
PC	Protection Component
PDP	Policy Decision Point
PEP	Policy Enforcement Point
PO	Protection Orchestrator
REST	Representational State Transfer, a stateless, client-server, cacheable communications protocol
SAML	Security Assertion Markup Language
SC	Secure Computation
SHE	Somewhat Homomorphic Encryption

SOA	Service-Oriented Architecture
SSL	Secure Sockets Layer, a cryptographic protocol for securing network connections
SSP	Secure Signal Processing
STR	Short Tandem Repeat, a tract of repetitive DNA
TOSCA	Topology and Orchestration Specification for Cloud Applications
VCF	Variant Call Format, a standard for storing gene sequence variations
VM	Virtual Machine

1 Introduction

Research in WITDOM focuses both on the development of efficient and practical techniques that can be used to securely and privately process data by Information and Communication Technologies (ICT) services owned by third-party providers of distributed processing and storage, as well as on the development and deployment of a private and secure-by-design framework which aggregates the developed techniques.

The architectural work package, WP4, aims at defining a flexible end-to-end secure architecture which forms the basis for securing use-case applications addressed in the project, e.g., use-case related to e-Health and financial scenarios. In particular, as an extension of the generic architecture proposed in Deliverable D4.2 this deliverable presents an adaptation layer to enable the WITDOM architecture to use in cloud environments. This enables the adoption of WITDOM when the untrusted domain is a cloud environment and thus allows services protected with WITDOM to take advantages from cloud benefits, such as, scaling resources and cost efficiency.

1.1 Purpose of the document

This report describes the final specification of an adaptation layer for the generic architecture proposed in Deliverable D4.2 and its adoptions to e-Health and financial scenarios to cloud computing environments. It describes how the components of the architecture need to be adapted in order to work in a hybrid cloud model, and how they can be provisioned, managed and deployed in a way that eases the work of developers in the sense of automation and integration. Furthermore, it also describes how the adapted architecture benefits from big data mechanisms that allow WITDOM protection components (e.g., Anonymization) to process large volumes of data. Additionally, the document shows how cloud-based computing improves the WITDOM use-cases, provides implementation guidelines corresponding to the cloud scenario, and examines a list of architectural challenges. Deliverable D4.6 is the final outcome of the following task, in accordance with the Description of Action (DoA).

- *Task 4.4:* Adaptation to Cloud-based environments.

1.2 Relation to other project work

WP4 connects the research results of WITDOM from WP3 (Basic Research on Enabling Privacy and Cryptographic Tools) to the platform and prototypes of WP5 (Privacy-Preserving Platform Toolkit and Prototypes). In this sense the architecture designed in WP4 demonstrates the role of the research technology in the actual platform. Additionally, the architecture can be used to implement the requirements and functionalities gathered in WP2 (Requirements Analysis and Prototypes Evaluation), more specifically in Deliverable D2.1 and D2.2, in combination with the relevant legal, ethical and societal issues identified in WP6 (Legal Requirements and Validation). A complete list of requirements that are fulfilled by WITDOM components can be found in Deliverable D3.2.

The outcome of this deliverable is related to work developed in Task 4.1 (Architecture and design of security, privacy and verifiability technology), Task 4.2 (Definition and design of system architecture for e-Health scenario), and Task 4.3 (Definition and design of system architecture for financial scenario). This deliverable completes the preliminary specification of the adaptation layer for cloud computing which was presented in Deliverable D4.5. In particular, the adaptation layer has been updated and finalized taking into account the recent developments of the prototypes in WP5.

1.3 Structure of the document

This document continues in Section 2 with a review of different cloud services and deployment models and discusses which are suitable for WITDOM. An overview of the adaptation layer of the generic WITDOM architecture to cloud environments is provided in Section 3. There, the deployment system chosen by WITDOM as well as a detailed description of how the Core Components and Protection Components need to be adapted to cloud environments is provided. Section 4 shows how cloud-based computing improves operations related to

individual e-Health and financial scenarios, and Section 5 provides implementation guidelines corresponding to the cloud adaptation. The document ends with a discussion on a range of architectural challenges in Section 6: from management to technical challenges foreseen for the use of the Cloud.

2 Cloud services and deployment models

This section gives an overview of common cloud service types (Section 2.1) and deployment models (Section 2.2) and discusses which are suitable for WITDOM. This is followed by an introduction of different cloud platforms (Section 2.3) which represent an example of modern cloud technology. In particular, we focus on two cloud platforms, namely OpenStack and Amazon AWS, which are used for the WITDOM prototypes in WP5.

2.1 Cloud services

Cloud computing is a broad term that describes a wide range of services. There are several characteristics that are essential for cloud computing, for example, the ability to receive services without long delays, the ability to access the services via many different platforms (desktop, laptop, mobile), the ability of the service to scale at demand peaks, and the ability to measure services and thus provide a metered billing.

Cloud computing is usually divided into three categories: Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS). Applications that are designed for end-users, like Facebook or Gmail, fall into the SaaS category. Applications that offer tools and services to make developing and deploying of end-user applications easier, like Google App Engine or Cloud Foundry, are categorized as PaaS. The IaaS model comprises servers, storage, networks, operating systems, and software for managing virtual machines, for example, Amazon Web Services (AWS), IBM Bluemix, or Google compute Engine (GCE).

The WITDOM framework fits into the PaaS model as it offers tools for end-to-end (E2E) protection of data and services (application) residing in an untrusted infrastructure. That is, end-users are not consuming WITDOM tools directly, instead they are integrated by service developers to protect their services according to their security requirements. This approach is completely transparent for the end-user. E2E means that data when stored in the cloud cannot be accessed or understood by attackers or rogue system administrators. Better said, whenever the data stored is accessed by non-authorized stakeholders (e.g., honest but curious cloud providers, attackers or rogue cloud provider insiders) only minimum personal data is revealed.

Traditional security methods, such as encryption at rest, usually prevent any processing over the protected data. Other methods, such as anonymization or data masking do allow (up to some extent) the processing of the data, but usually limit the utility of the processes results. WITDOM is applying a novel cloud model, which enables processing of end-to-end protected data on the cloud computing provider's servers that are not necessarily trusted. WITDOM achieves this by preprocessing data in the trusted domain (to protect the data) and then moving the preprocessed and protected data to the untrusted domain (cloud computing provider) where the main processing of data takes place.

2.2 Deployment models

A cloud deployment model specifies the type of cloud environment, in particular the ownership of the infrastructure, its size, and the access control. Three deployment models are common; *public cloud*, *private cloud*, and *hybrid cloud*. In the following we present descriptions of these models as defined by NIST[1].

A *private cloud* may be owned, managed, and operated by the organization, a third party, or some combination of them, where the infrastructure is provisioned for exclusive use by a single organization comprising one or more consumers (e.g., business units). Organizations maintain their own physical servers, network and storage infrastructure hosted on or off premises. The critical criteria for a private cloud is the fact that is under full and exclusive control of its owner. For certain use-cases this is an important security criteria. Since private clouds

[1] http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf

are typically deployed on the organization's own on- premise data center, the availability and resilience of this infrastructure depends on decisions made with the IT team of the organization. This also includes the responsibility for maintenance for hardware and software systems. Although the physical resources of a private cloud are kept by cloud's owner, which is also it's customer, a private cloud enjoys features that are common to any cloud environment, such as: automation (resources do not need to be manually provisioned or decommissioned), and scalability (increasing storage and computing resources should add no management overhead). An example of a cloud computing software that can be used to deploy private clouds is OpenStack presented in Section 2.3.1.

A *public cloud* can be owned, managed, and operated by a business, academic, or government organization, or some combination of them, and are provisioned for open use by the general public. The physical resources for computing and storage are offered by the cloud provider in a way that organizations can simply connect to the cloud and use resources on a pay-per-use basis. Typically, the physical infrastructure is shared among multiple customers of the cloud provider and logical separated leveraging virtualization layers and access control. This also means that for security critical applications the exclusive access to resources may not be guaranteed since the cloud provider has always root access to all resources. However, this model gives organizations the advantage to avoid capital investment on their own on-premises infrastructure, and instantly scale the infrastructure up or down according to their needs. Another factor that makes public clouds very economical is that the public cloud provider is responsible for the maintenance of the cloud and of its IT resources. An example of public cloud is Amazon Web Services (AWS), introduced in Section 2.3.2, which offers cloud computing services that operate in several geographical regions around the world, which enables developers or architects to deploy a high-availability solution where the failure of the public cloud's infrastructure in one region does not take the entire system down. Among the services offered by Amazon AWS, we can mention computing, networking, storage and analytics.

A *hybrid cloud* is a composition of two or more distinct cloud infrastructures (e.g., private and public) that remain unique entities, but are bound together by standardized or proprietary technology that enables data and application portability (e.g., cloud bursting for load balancing between clouds). This model combines the benefits of the private and public cloud model, for instance, it provides the security guarantees of a private cloud and extends it with the flexibility and scalability of a public cloud. In order to maximize the benefits of running a hybrid cloud and minimize management costs, a cloud management platform is required.

The hybrid cloud model is a popular deployment model and widely used among small to large enterprises. In particular, business critical services and processes are often deployed in the private infrastructure because of high security requirements, wheres non-sensitive data and services are outsourced to the public cloud. The WITDOM platform fits perfectly in this deployment model but pushes the security limitations beyond. That is, by leveraging WITDOM it allows to outsource also business critical services and sensitive data to the public cloud. Some WITDOM services are running in the private cloud (data transformation, data protection, preprocessing), but in order to scale out and conduct main processing of the data on external resources, public cloud is used additionally. Consequently, we choose the hybrid cloud model as the preferred cloud model for WITDOM. Section 3 gives a detailed description of the cloud adaptation layer for the WITDOM platform.

2.3 Cloud platforms

We complete this section by introducing OpenStack (Section 2.3.1, a private cloud platform, and two popular public cloud service providers, namely Amazon Web Service (Section 2.3.2) and IBM Bluemix (Section 2.3.3). OpenStack and Amazon web services are used for the WITDOM prototypes in WP5. We also introduce IBM Bluemix as an alternative public cloud provider, however, WITDOM is not restricted to the presented platforms.

2.3.1 OpenStack

OpenStack is an open-source cloud computing platform that is actively developed by a large community of developers. Initially started with only a small set of services for computing, network, and storage, OpenStack now provides a rich set of features through a variety of complementary services. Each service offers an API that

allows interaction among the services and with the users. It is highly scalable and can be deployed on only a few nodes or in a data center with hundreds or thousands of nodes in a cluster. Figure 1 depicts the OpenStack architecture containing its main services and their relation. Due to the ongoing development of OpenStack there exist more services today than shown in the figure. However, in this deliverable we focus only on the basic services which are currently stated as stable. Table 1 gives a brief description of each OpenStack core service whereas Table 2 gives a brief description of optional OpenStack services which are relevant for the WITDOM platform. More detailed information about OpenStack and its services can be found on the OpenStack website[2].

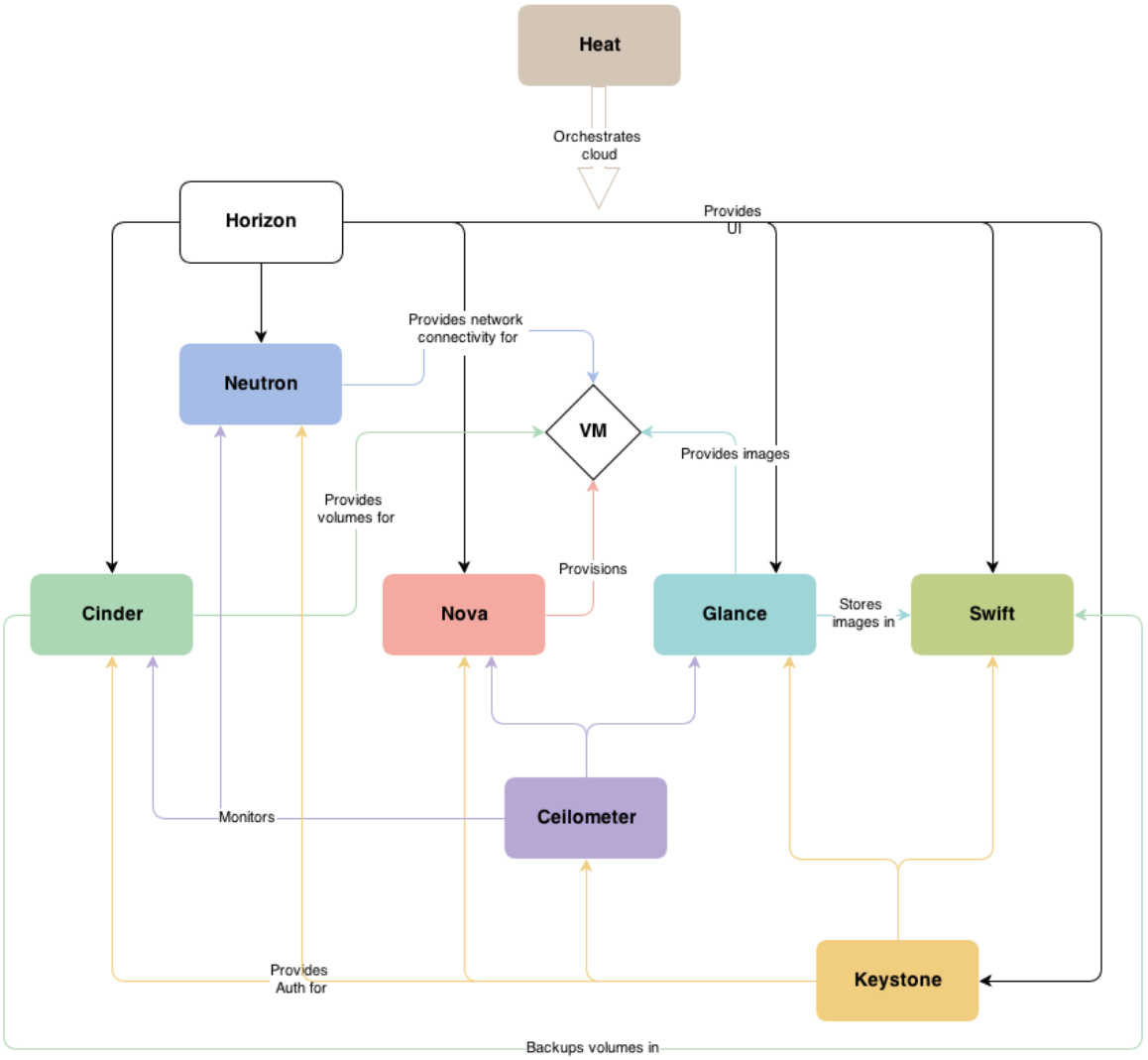

Figure 1: OpenStack architecture. Graphic source: `https://www.openstack.org/`

Service	Description
Nova (Compute)	It is the core service of OpenStack that manages virtual machines (VMs) in the infrastructure. It is responsible to start-up, scheduling and decommissioning VMs.
Neutron (Networking)	Provides Network-as-a-Service for OpenStack services. It allows users to define networks and integration in virtual machines.

[2]OpenStack website – `https://www.openstack.org`

Swift (Object storage)	It is a highly available, distributed, eventually consistent object store that allows users to store arbitrary unstructured data via a HTTP based API. Data objects are replicated within the service and thus fault tolerant. Other OpenStack services, such as Glance, also rely on Swift.
Cinder (Block storage)	It provides persistent block storage to virtual machines. It allows users the creation and management of block storage devices.
Keystone (Identity service)	It provides authentication and authorization service for OpenStack services. It also provides a catalog of all available services endpoints of the OpenStack infrastructure deployment. As discussed in Deliverable D4.2, this service matches the architecture of the WITDOM IAM component and the associated end-user requirements, and will therefor be used for its implementation.
Glance (Image service)	Provides a virtual machine disk image storage services. It allows users to store and retrieve disk images. Nova uses Glance during VM provisioning.

Table 1: List of OpenStack core services.

Service	Description
Horizon (Dashboard)	It provides a web-based user interface to interact with OpenStack services, such as launching virtual machines, configuring network and access control.
Ceilometer (Telemetry)	Provides monitoring services for other OpenStack services. It allows users to meter different measurements of the infrastructure and therefore conduct benchmarking, billing or other system evaluation.
Heat (Orchestration)	It provides an infrastructure orchestration services that allows users to define complex distributed cloud applications using a composition of multiple services.
Barbican (Key manager)	It provides secure storage, provisioning, and management of cryptographic keys and certificates. Furthermore, it interacts with Certificate Authorities to issue, renew, and revoke certificates. As discussed in Deliverable D4.2, this service matches the architecture of the WITDOM KM component and the associated end-user requirements, and will therefor be used for its implementation.
Sahara (Big data service)	This component allows processing data in large scale. Sahara provides a platform for big data analytics, allowing distributed and parallel data processing of data. It works well with Hadoop and Spark, which are an open-source software libraries which distribute storage and processing of large data sets across groups of servers.

Table 2: List of OpenStack optional services.

The central feature of OpenStack is to allow users to create an infrastructure of *virtual machines (VMs)* which are connected via a virtual network. Users can launch new VMs and configure virtual resources such as virtual CPUs, memory, and disk space using a web-based user interface, the so called *dashboard service (Horizon)*. The deployment of VMs is completely transparent for the user, that is, the *compute service (Nova)* decides on which hypervisor node a new VM is started according to the allocated virtual resources. During the provisioning

processes the compute service interacts with other OpenStack services. The *image service (Glance)* provides a system image (e.g., Ubuntu cloud image or images customized by the user). These images are stored in the *object storage service (Swift)*. Swift can also be accessed by users directly to store any application data. The *block storage service (Cinder)* provides a disk volume to a VM according to the user requirements. Cinder also uses Swift to back up volumes. The network connection of VMs is managed by the *networking service (Neutron)*. Neutron assigns network addresses to the network interfaces of the VMs according to a specified network topology and with respect to firewall settings. There exists an *identity service (Keystone)* that enforces identity and access management in OpenStack. Keystone is responsible to authenticate users and services among each other when accessing resources. We refer to Section 3.3.1 for a detailed description WITDOM's identity and access management component. The *telemetry service (Ceilometer)* provides monitoring functionality for the infrastructure; for instance, users can monitor their system usage, such as CPU, disk and network utilization for benchmarking, and Ceilometer can be used for billing. Users can orchestrate their infrastructure, including VM provisioning and network configuration using the *orchestration service (Heat)*. Users simply describe their infrastructure in a template file according to their application requirements and delegate the creation to Heat. The *key management service (Barbican)* provides secure storage, provisioning and management of cryptographic keys and certificates. Section 3.3.2 elaborates the key management implemented by WITDOM. Finally, OpenStack provides the *big data service (Sahara)* that allows processing data in large scale. Sahara enables users to fast provision and manage of data processing clusters (based on Hadoop or Spark) on OpenStack. It may be used as a cluster manager of WITDOM's big data component that is described in Section 3.3.7.

There are basically two possible approaches to use OpenStack to deploy WITDOM components in the trusted domain. First, WITDOM components can be developed as individual OpenStack services using predefined service specifications by the OpenStack community. This allows the best possible integration of WITDOM into the OpenStack cloud platform. On the other hand, this may also bind WITDOM too closely to OpenStack. Therefore, we recommend to apply the second approach, where we host WITDOM components, such as the broker, the protection orchestrator, and the protection components, as microservices inside virtual machines or containers within OpenStack. This provides the flexibility to reuse WITDOM components and change the underlying cloud platform depending on user requirements.

2.3.2 Amazon Web Services

Amazon Web Services (AWS) is a commercial cloud platform that offers on-demand computing resources and services through all cloud service models, namely IaaS (for example, EC2[3]), PaaS (for example, AWS Lambda[4]), and SaaS (for example, Amazon WorkMail[5]). Officially launched in 2006, it offers a *pay-as-you-go* billing model where users pay only for what they have used, for instance, computation time by the hour or data volume for storage and transfer-related services. This allows flexible usage of the scalable infrastructure voiding unnecessary investments in on-premise resources. AWS provides a global physical infrastructure spread over different geographic locations that enables fault-tolerant, highly available, and scalable services to the users. Amazon offers a broad range of services, such as for computing, storage or analytics. Certainly, the most popular services are *Amazon Elastic Compute Cloud (EC2)* and *Amazon Simple Storage Service (S3)*. Moreover, Amazon offers services for computing, network, storage and content delivery, databases, deployment management, and application services. A full list of available services provided by AWS can be found on the AWS website[6]. Figure 2 briefly illustrates the AWS platform.

WITDOM components residing in the untrusted domain, such as protection components, the broker, and application services can be deployed as microservices in virtual machines hosted on EC2. Application services may integrate other AWS services, such as S3, to store data applying protection mechanisms provided by WITDOM

[3] https://aws.amazon.com/ec2/
[4] https://aws.amazon.com/lambda/
[5] https://aws.amazon.com/workmail/
[6] https://aws.amazon.com/

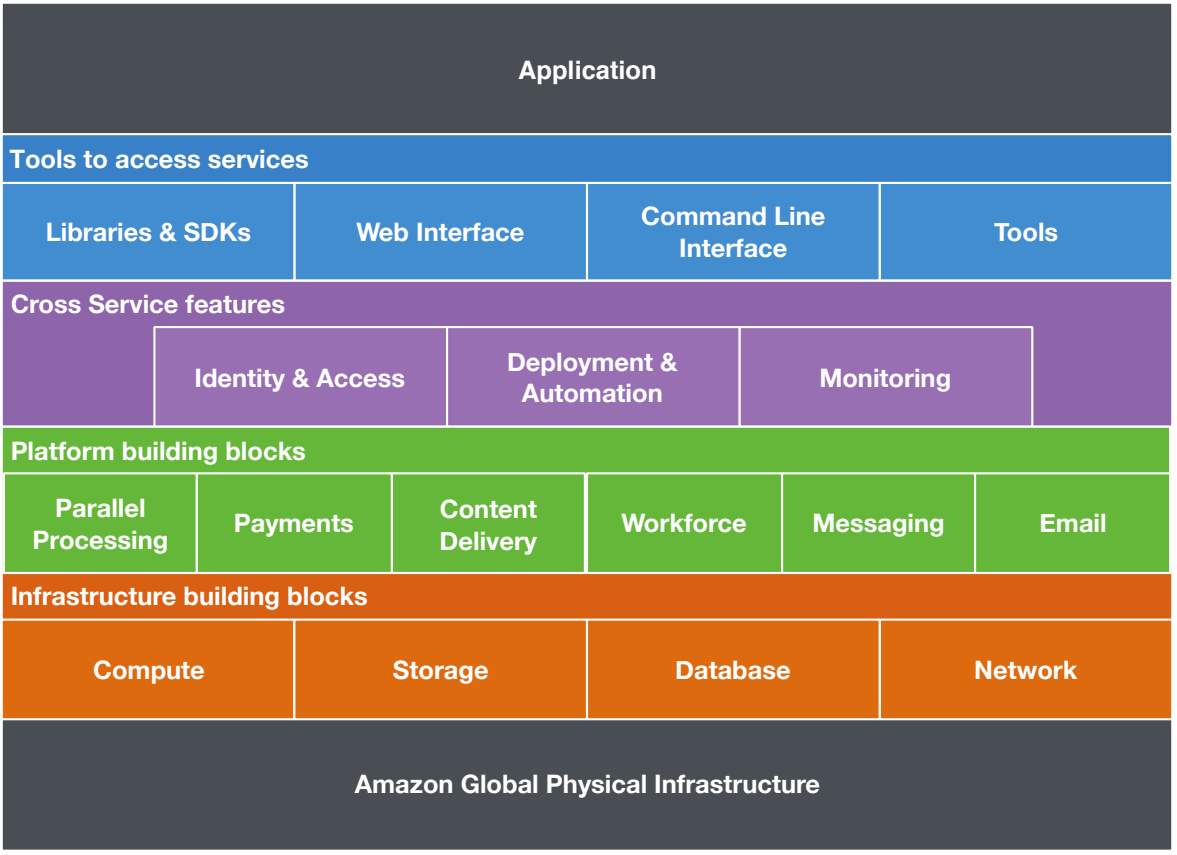

Figure 2: Amazon web services platform. Graphic based on `https://aws.amazon.com/`

protection components. However, WITDOM platform is not bound to Amazon services therefore alternatively IBM Bluemix, as discussed in the next section, may also be used as public cloud platform.

2.3.3 IBM Bluemix

IBM Bluemix [7] is a cloud computing platform that combines platform as a service (PaaS) with infrastructure as a service (IaaS). Additionally, Bluemix has a rich catalog of cloud services that can be easily integrated with PaaS and IaaS to build business applications rapidly.

All IBM cloud resources that are deployed in public and dedicated environments are hosted from IBM Cloud Data Center locations around the world. IBM Cloud Data Centers provide regional redundancy, a global network backbone connecting all data centers and points of presence, and stringent security controls and reporting. Bluemix infrastructure is one platform, which takes data centers around the world that are full of the widest range of cloud computing options, then integrates and automates everything. IBM Cloud Data Centers are filled with first class computing, storage, and networking gear. Each location is built, outfitted, and operated in the same way, that is, exactly the same capabilities and availability are present everywhere. Locations are connected by the industrys most advanced network-in-a-network, which integrates distinct public, private, and internal management networks to deliver lower total networking costs, better access, and higher speed. Also, the data centers and network share a single proprietary management system. One management tool allows to control everything–every bare metal server, virtual server, and storage device–all accessible by API, portal, and mobile applications.

Bluemix infrastructure offers powerful bare metal servers and flexible virtual servers in a single seamless

[7] `https://www.ibm.com/cloud-computing/bluemix/`

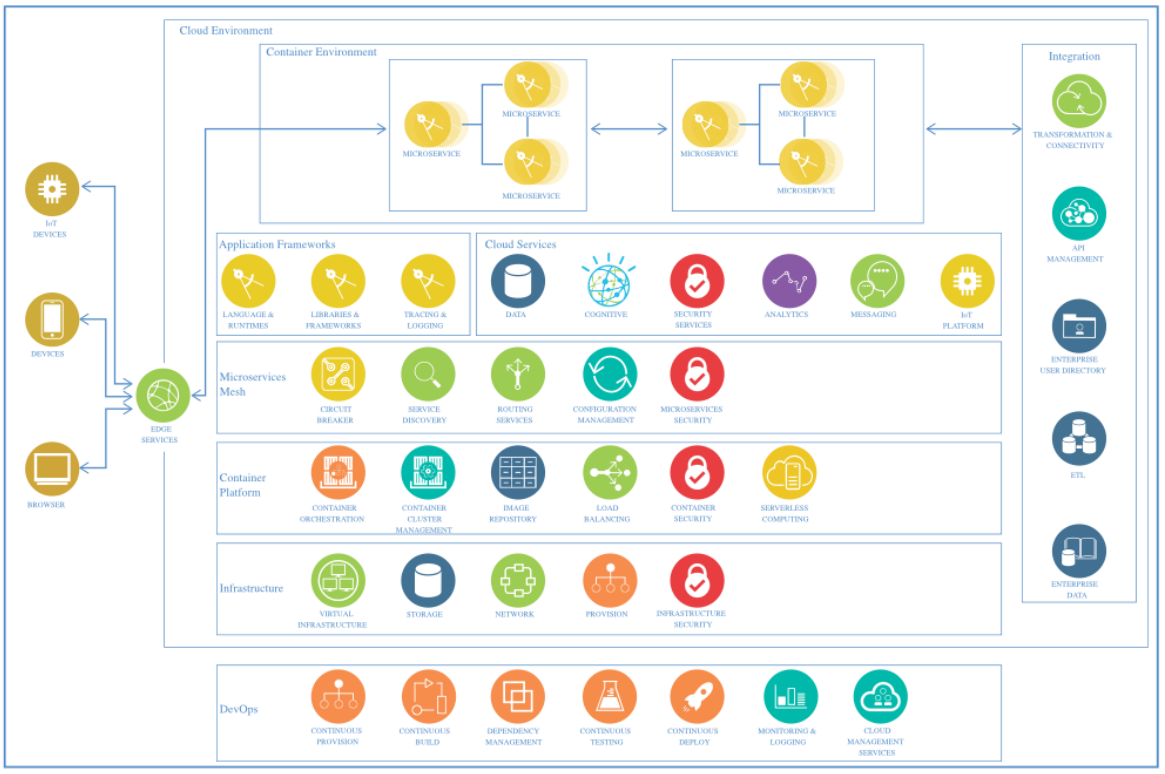

Figure 3: IBM Bluemix platform consists of a large number of different components and services, such as, infrastructure, container platform, mircoservices, application frameworks and cloud services. Graphic source: `https://www.ibm.com/devops/method/content/think/practice_get_started_with_architectures/`

platform. All are provided on demand and billed on monthly or hourly terms. Bare metal servers provide the raw horsepower for processor-intensive and disk I/O-intensive workloads and can be configured to exact specifications. Virtual servers allow for high speed of deployment, flexible scalability, and pay-as-you-go billing. For high performance computing, graphics processing unit (GPU) servers can boost performance according to customers requirements, available by the hour or monthly.

Bluemix infrastructure offerings are connected to a three-tiered network, segmenting public, private, and management traffic. Infrastructure on a customer's Bluemix account might transfer data between such infrastructure across the private network at no cost. Infrastructure offerings, such as bare metal servers, virtual servers, and cloud storage, connect to other applications and services in the Bluemix catalog, such as Watson services, containers, or runtimes, across the public network. Data transfer between those two types of offerings is metered and charged at standard public network bandwidth rates.

The Bluemix catalog [8] offers a wide range of application services, such as, for data analytics, blockchain, Internet of things, mobile, and Watson. IBM Bluemix services and infrastructure allow developers to build, deploy and scale their application according to their individual requirements.

3 Adaptation of the WITDOM architecture to the cloud

3.1 Overview

This chapter describes how the WITDOM generic architecture presented in Deliverable D4.2 is adapted to cloud environments.

The cloud adaptation of WITDOM's generic architecture enables the deployment of all generic architectural components in cloud environments. In particular, the adapted architecture assembles several building blocks for enhancing the security of applications and organizes a number of protection components in such a way that applications can access them in a composable and modular way. The cloud adaptation allows to instantiate the WITDOM platform on an elastic infrastructure that requires only minimal management effort and also benefits from other cloud advantages such as cost efficiency in terms of storage and computing power.

A typical instantiation of WITDOM in a cloud infrastructure uses the hybrid cloud model. All WITDOM components previously placed in the trusted domain are deployed using an open-source private cloud, such as OpenStack, and the components in the untrusted domain can be deployed in a public cloud, for instance using Amazon AWS, IBM Bluemix, or any other cloud platfrom. This means that WITDOM uses the private cloud to protect sensitive data and run some in-house services, and uses the public cloud to outsource applications to run, for instance, heavy data analyses. The private cloud is implemented in the WITDOM's user own data center, thus enjoying intrinsic on-premise security levels. Additionally, the cloud allows for flexibility of provision, meaning that resources can be assigned to applications on-demand, and other advanced management tools to avoid lengthy IT intervention. The public cloud permits the highest level of efficiency and shared resources, enabling on-demand access to storage and computing power. Figure 4 illustrates the adaptation of WITDOM's generic architecture to the cloud. Note that the design of the cloud adaptation is generic that it can be easily instantiated on other cloud platforms than OpenStack, Amazon, and IBM Bluemix.

The components in the hybrid cloud used by WITDOM are provisioned and managed by *Cloudify*[9], an open-source cloud orchestrator software. Cloudify allows WITDOM developers and system administrators to automate the deployment of components of the architecture; it simplifies the communication among components and connections to applications. Cloudify is also used for post-deployment operations, for example to allocate or deallocate server instances in the cloud when necessary. More details about the WITDOM deployment system is provided in Section 3.2.

Using the cloud-based WITDOM architecture, all *applications* that run on behalf of end-users are hosted in the trusted domain, and communicate with the private cloud to request services. Applications are deployed in the trusted domain and may benefit from application-specific *services* hosted in the private cloud (also part of the trusted domain). Applications access particular security-critical functions and services through a *broker* component, which mediates their interaction with the infrastructure. Services may be provided by components in the trusted domain or by others in the untrusted domain. Application- specific services running in the untrusted domain are called *secured services* whenever they satisfy the protection requirements of the end-user. That is, using a secured services requires some protection logic before and after the actual business-related processing.

Services correspond to processes that take a specific business role for the end-users, for example, a FASTQ sequence alignment process or artificial neural network training, according to Deliverable D2.2. The services residing in the untrusted domain require certain security measurements for protecting the data and algorithms with which they operate. The *protection components (PC)* in the architecture are responsible for providing techniques that guard these assets from attacks whenever outsourced to the untrusted domain.

[8] `https://console.bluemix.net/catalog/`
[9] `http://getcloudify.org/`

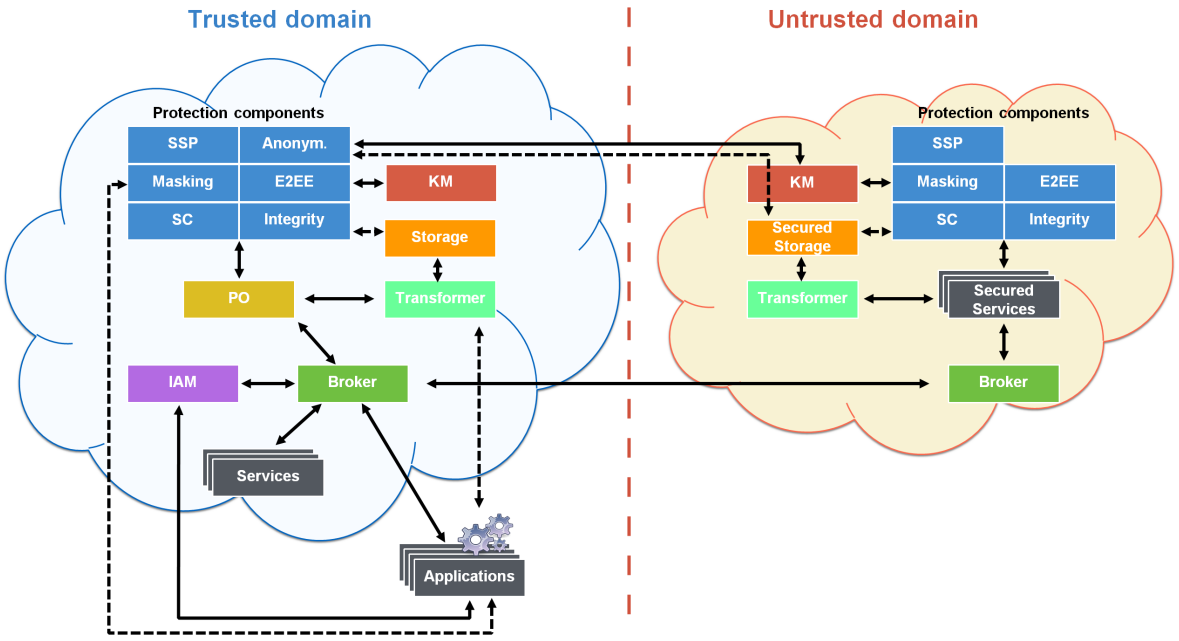

Figure 4: Simplified view of adaptation of the WITDOM architecture for cloud computing.

For the sake of portability, and to allow continuous integration, PCs are placed in *software containers*. Containers allow developers to package a software and everything it needs to run, such as libraries and dependencies, in a single package that always runs the same way independent of the environment. A container shares the kernel of an operating system with several other containers. Since containers do not package operating systems, they are extremely light-weight. This means that the developer does not need to install the software dependencies and this increases flexibility and portability. WITDOM uses *Docker*[10], an open-source software containerization platform to package the PCs.

Protection components only process data that are in a common WITDOM format, stored in standard SQL-based databases. Hence, data coming from applications' data-stores are first, if necessary, converted into the WITDOM format via a *transformer* component, which is able to transform any data that have the format of data used by registered services into WITDOM format. Large data processing in the trusted domain can be enabled by the use of a *big data component* that uses distributed and parallel data processing technique for efficiency. WITDOM leverages Spark[11] to implement this component and may integrate with Sahara[12] for cluster management. Alternatively other big data frameworks, such as Hadoop [13], may be used with WITDOM if required by the end-user.

By using Cloudify, the broker component knows the location of the various services and WITDOM components, their deployment, and how to access them. In particular, Cloudify maintains a list of addresses of the hosts system of each WITDOM component that also includes information whether a hosts system is located in the trusted environment (private cloud) or in the untrusted environment (public cloud). All services, whether in the trusted or untrusted domain, are registered with the broker. The broker is also concerned with matching the security preferences of an application to the security features offered by the services and protection components. It decides, upon receiving a request from an application and in accordance with *security preferences*, whether the request should be handled in the trusted or untrusted domain. The broker consists of two elements, one running in the trusted domain and the other in the untrusted domain. These components communicate with each other

[10] https://www.docker.com/
[11] https://spark.apache.org/
[12] https://docs.openstack.org/sahara/latest/
[13] https://hadoop.apache.org/

through the establishment of a mutually authenticated secure channel using SSH/TLS.

An *identity and access manager (IAM)* component providing authentication and authorization services for end-user applications and individuals is deployed using OpenStack Keystone. The IAM mainly interacts with the other components through the broker, but they may also reach out to IAM directly, whenever there is a need for service-specific access control and authorizations. OpenStack Keystone offers token-based authentication based on credentials provided by the end-user applications. Authentication works as the following. An end-user application obtains a token for a certain service by providing credentials to the IAM. The token is sent along with a service request to the broker which verifies the token and forwards the request accordingly.

A *protection orchestrator (PO)* mediates between the applications, the broker and the *protection components (PCs)*, which introduce the actual protection measures, for example, by using cryptographic techniques such as masking or encryption. The PO coordinates the application of multiple different kinds of protection relying in a dynamic configured set of protection components using an application specific *protection configuration*.

The WITDOM architecture also offers a *key management (KM)* component which be used to generate and store cryptographic keys that may need to be used by different PCs, users, and also keys needed to establish secure channels. The KM component is instantiated with OpenStack Barbican, which can provision keys, certificates and other cryptographic material, and interacts with certificate authorities in order to issue, renew, or certificate keys related to users and WITDOM components.

Taken together, this design offers not only several benefits of a Service-oriented Architecture (SoA), namely extensibility, service re-use, flexibility, and loose coupling, but it also enjoys the advantages of cloud computing such as easy deployment and post-deployment, elasticity (in terms of on-demand allocation of resources) and cost efficiency. In summary, the adaption layer for cloud computing of the WITDOM architecture enables:

- A complex service to be composed as successive calls to individual services (disregarding whether they are deployed in the trusted or untrusted environment).
- A service to be transparently (to the end-user) moved from the trusted domain to the untrusted domain (only affecting performance, utility and privacy levels).
- A service to be provided in both domains, and the domain where it is finally executed may depend on rules or on parameters in the service call.
- End-users to develop their own applications that make use of one or several of the services available without knowing anything of secure processing or underlying protection techniques. For instance, a genomic analysis application might call the alignment algorithm or a variant annotation through the service interfaces offered by the WITDOM.
- Reduction of capital investment for the whole SaaS infrastructure as provided by a public cloud provider.
- Easy deployment and post-deployment of the WITDOM components using commodity open-source tools.
- Automation of infrastructure management, for best use of resources.
- Easy testing and integration of end-user application with WITDOM components.

The following subsections gives more details of how the WITDOM architecture can be deployed in the cloud.

3.2 Deployment system

To make WITDOM solution easily adoptable for its users, a simple mechanism needs to be provided to deploy its components. That counts for components in trusted as well as components in the untrusted domain. Furthermore, WITDOM needs to be able to operate in different environments and on the infrastructure of different cloud service providers.

To achieve this, Cloudify has been adopted. Cloudify is an open-source platform that automates the installation and deployment of cloud applications. It provides support for several cloud providers including OpenStack[14], AWS[15], CloudStack[16], Microsoft Azure[17], Rackspace[18], and VMWare[19]. Although OpenStack contains an Orchestration component, called Heat[20], this component works exclusively with OpenStack, and does not support a multi-cloud infrastructure, unlike Cloudify.

WITDOM uses Cloudify for deployment of components in both trusted and untrusted domain. This way the WITDOM trusted components is deployable on a wide range of private clouds (being this OpenStack, CloudStack, VMWare or others) and WITDOM untrusted components is enabled for most of the main cloud infrastructure providers (being this Amazon AWS, IBM Bluemux, Microsoft Azure, Rackspace and others).

Despite Cloudify being an essential part of WITDOM deployment model, interaction with Cloudify is obscured from the administrator as much as possible. For this reason, a component called Dashboard is being developed, which provides a user interface for easier management of WITDOM components. The WITDOM administrator only needs to deploy the WITDOM Dashboard and provide credentials for the public cloud. The Dashboard then automatically installs and configures all components in the private and public cloud as previously chosen by the administrator.

For the actual deployment of the WITDOM platform, we use a combination of Cloudify and a configuration management tool as discussed in Section 3.2.1. However, in order to simplify the integration process and to additionally enable the integration of WITDOM with platforms like CoreOS[21] and Kubernetes[22], we use a container approach, as discussed in Section 3.2.2.

3.2.1 Cloudify deployment using configuration management tools

To enable WITDOM components to be deployable by Cloudify through the WITDOM Dashboard, for each component a set of instructions to get the component up and running needs to be prepared and a Cloudify application blueprint[23] needs to be provided in TOSCA format. The blueprint contains logical description of a WITDOM component in terms of nodes (for instance, a server node and a database node) and how they are related to cloud provider's infrastructure.

To demonstrate how to prepare a component deployment instructions, an example of a provisioning script in the form of a Cloudify application blueprint for the OpenStack Barbican[24] (blueprints vary depending on the chosen cloud provider), which implements the WITDOM Key management component), are included in Annex A.

The provisioning script installs necessary software libraries required by Barbican, pulls Barbican source code from Git repository and installs it inside a Python virtual environment. Then, it copies the configuration file to the node we are provisioning and starts the Barbican service.

The blueprint is used to enable Cloudify to create the appropriate cloud infrastructure - a single virtual machine instance, to which it assigns a chosen security group and a floating IP. With the blueprint we also instruct Cloudify to instal and configure the component on the newly created Barbican virtual machine. When Cloudify deployment has finished, the Dashboard is able to retrieve some basic information about Barbican deployment by inspecting deployment ouptputs, specified in the last block of the Cloudify blueprint. This information can be later propagated to other parts of the WITDOM architecture to locate individual WITDOM components (in our example, Barbican service endpoint).

[14] https://www.openstack.org/
[15] https://aws.amazon.com/
[16] https://cloudstack.apache.org/
[17] https://azure.microsoft.com/en-us/
[18] https://www.rackspace.com/
[19] http://www.vmware.com/
[20] https://wiki.openstack.org/wiki/Heat
[21] https://coreos.com/
[22] http://kubernetes.io/
[23] http://getcloudify.org/guide/3.0/understanding-blueprints.html
[24] https://wiki.openstack.org/wiki/Barbican

3.2.2 Deployment using Docker

Docker is a software containerization platform. It automates the deployment of applications in software containers. Containers allow developers to package up an application with all of the parts it needs. This way, the application can run on any other Linux machine.

Docker can be seen as a lightweight virtual machine. However, it does not create a whole virtual operating system, but uses the host operating system kernel and a layered filesystem where common files are shared amongst containers.

WITDOM primarily uses Cloudify paired with Dockerfiles[25]. Dockerfile is a Docker descriptor where all libraries and configurations that are needed for an application/component are specified. Being able to deploy WITDOM components via Dockerfiles enables integration testing of the platform on one computer. This greatly simplifies the integration process since it avoids a heavy infrastructure of the distributed architecture (WITDOM components are deployed in two different clouds - private and public). Thus, during the integration within the project, all WITDOM components are deployed on a local computer where each component exposes its API endpoint (the communication between the components is the same as in the distributed environment, only the endpoint URLs are local). However, in a production deployment of WITDOM where more computational resources are needed, WITDOM components can be deployed on multiple host systems.

In annex B, we present a Dockerfile, which installs everything needed for OpenStack services Barbican (used as the WITDOM Key management component) and Keystone (used as the WITDOM Identity and access management component). At the end of the file, there are three ports exposed - one for the Keystone public endpoint, one for the Keystone admin endpoint, and one for the Barbican endpoint. Components that need to interact with Barbican and Keystone, can access their services using these endpoints.

Not only does Docker enable easier integration and testing, it also enables integration of WITDOM with platforms like CoreOS and Kubernetes. CoreOS is a lightweight Linux operating system made for clustered deployments providing automation, security, and scalability for applications. CoreOS makes large, scalable deployments simple to manage. It maintains a lightweight host system and uses containers for shipping the applications. Kubernetes is a container management software, which enables service discovery, automatic load-balancing, container replication etc. Kubernetes handles the entire life-cycle of a containerized application, including deployment and scaling. Thus, having Dockerfiles for WITDOM components enables deployment of the WITDOM platform on other systems than Cloudify as well.

3.3 Core components

3.3.1 Identity and access management

In WITDOM, the IAM component comprises two different modules and is thus implemented with two different open-source tools. Namely, the IAM core module, which is a standalone component that oversees identity and access management, takes advantage of OpenStack Keystone. The Auditing module responsible for access monitoring and logging, which is part of the Broker component, uses a node.js-based audit logging toolkit. Both components (further discussed in the following) are cloud-native and hence need no cloud-specific adaptation.

Authentication and authorization - OpenStack Keystone Keystone is an OpenStack project that provides identity, token, catalog and policy services for projects in the OpenStack family. However, it can be easily reused for other projects with distributed architecture (which can be run outside from an OpenStack architecture). Keystone supports token-based authentication and user-service authorization.

OpenStack Keystone supports integration with LDAP. This is crucial to meet WITDOM requirements related to connecting WITDOM IAM component to the existing IAM systems used by WITDOM users. However, connecting Keystone with LDAP, Active Directory or similar systems also enhances security since Keystone does

[25] `https://docs.docker.com/engine/reference/builder/`

not support some capabilities to improve security that are supported in these systems: password syntax and dictionary checking to prevent against weak passwords; password expiration; password history to prevent against password reuse; account lockout after some failed authentication attempts.

There are several requirements in WITDOM associated with integrating a strict authorization and authentication component that supports various level of rights and privileges (for example, requirements #68, #163 or #164 demand an IAM component with a fine-grained access policies and role-based access control functionality). This is provided by Keystone with a notion of a Role-Based Access Control (RBAC). By each call to the Broker, the information about the user (extracted from the token) will be checked. WITDOM specific set of roles and rules will be defined to meet the requirements.

Auditing - NPM audit-log toolkit In the previous iteration of this deliverable, we assumed that the enhanced version of the SPECS ConSec[26] component would be used for the auditing / logging functionalities in WITDOM. Afterwards, with the goal to simplify developments and integration, it has been decided to integrate some of the IAM's functionalities within the Broker component and thus use an open-source solution that is easy to integrate within its code. To this end, a node.js-based audit logging toolkit[27] has been chosen, which enables the Broker to save auditing logs of all requests that it receives. Further details are provided in a corresponding Section 3.3.3.

3.3.2 Key management

As introduced in Deliverable D4.2, the WITDOM Key management component is implemented with OpenStack Barbican.

OpenStack Barbican provides secure storage, provisioning, and management of secret data. It can handle different types of secrets, including symmetric keys, asymmetric keys, and raw secrets. It has a support for symmetric keys life-cycle management and for various backends for protection of keys, like Hardware Security Modules (HSMs). Furthermore, Barbican architecture, presented in Figure 5, has been designed to be highly scalable.

Some Protection Components and Core Components must need to retrieve secret data from Barbican services in order to obtain, for example, database credentials. Regarding the communication between these components and the Key Manager, WITDOM counts on a Java REST client named KM Client. It allows them to manage the different operations that are supported by Barbican (e.g., create secrets or containers, update the metadata of an existing secret, retrieve a secret's payload, maintain an access control list, etc.). This KM client can be integrated within the components as a JAR packaged library and non-java components would need a Java instructions wrapper or a Java client in order to use it.

There are no WITDOM specific extensions of Barbican needed for the cloud adaptation of the WITDOM architecture since it is already cloud-native. Within the project a general purpose Barbican client was developed that allows WITDOM components to integrate easily with Barbican.

3.3.3 Broker

The Broker is used as a secure gateway for routing the messages between other WITDOM components. It is deployed like the rest of the components, by means of the Cloudify orchestrator. Two instances of the Broker are deployed, one in the trusted and another in the untrusted domain. For this reason, each Broker instance has to be deployed by a different Cloudify instance, and then configured manually to establish a secure connection between them. The Broker deployment is specified with a Cloudify's TOSCA-standard blueprint.

The Broker has been designed to work in any infrastructure like a private or public cloud, or a mix of both. There are no changes in the deployment process, neither in the APIs or access control independently of the infrastructure where it is deployed. However, there are some constraints, specified in Deliverable D4.2, in order to have the Broker properly working.

[26] http://www.specs-project.eu/

[27] ttps://www.npmjs.com/package/audit-log

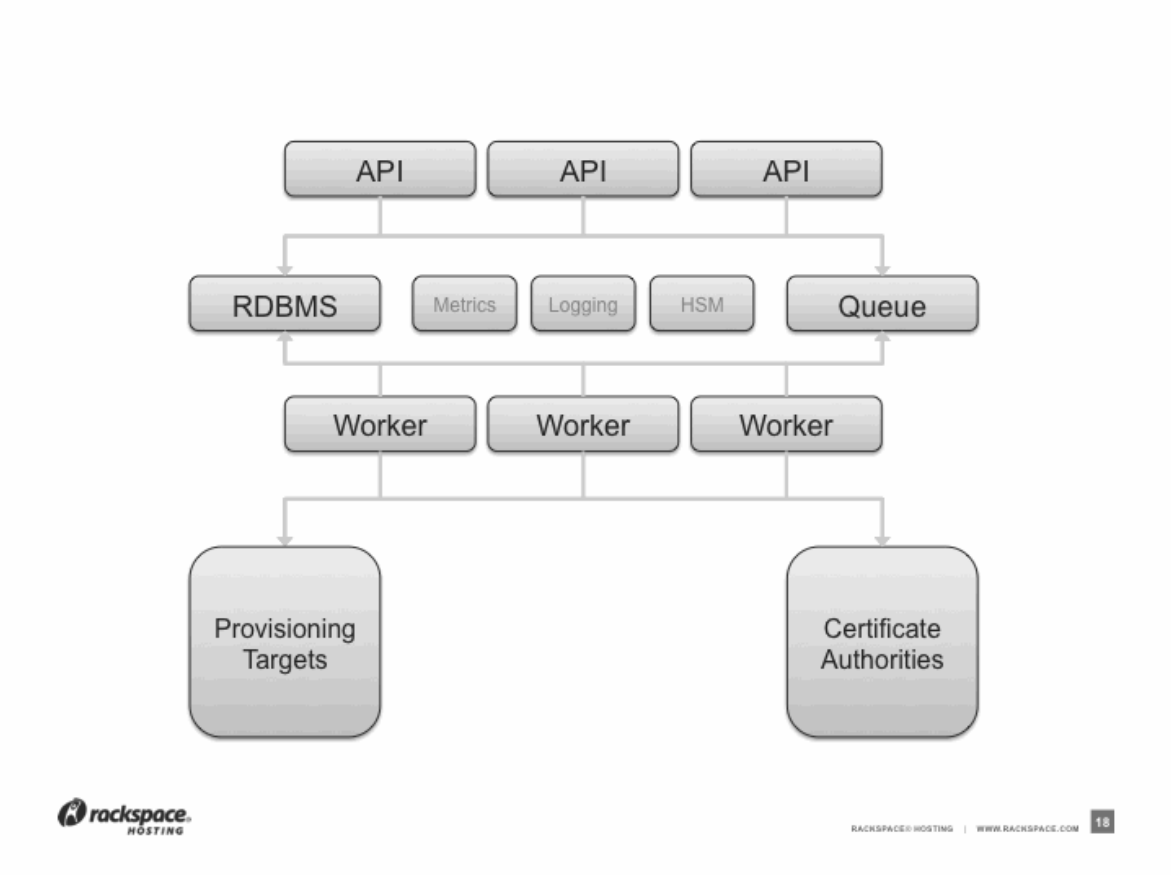

Figure 5: OpenStack Barbican architecture. Graphic source: `http://www.rackspace.com`

The Broker requires the following configuration:

- HTTPS port used to listen for connections.
- Domain's CA certificate in the Truststore.
- Certificate of the CA that signed the untrusted (trusted) domains' Brokers certificate in the Truststore (optional, only if there is at least one untrusted domain). This is used to validate the certificates in the mutual authentication process.
- Location of the untrusted (trusted) instance of the Broker.
- Location of the IAM component and its credentials.
- Location of Deployment Manager (Cloudify).
- Broker certificate in the Keystore.
- Internal components location (two separated MongoDB instances, one for storing the internal Broker data and the other for the audit module logs).

As the Broker routes all the requests generated in WITDOM (with the exception of the accesses to the storage components), it has the potential risk of becoming a bottleneck. In order to achieve the maximum scalability, each Broker instance (trusted and untrusted), which are symmetric by design, is splitted into its internal components. Figure 6 represents the changes that are necessary to deploy the Broker in a cloud environment, which typically

offers scalability, load balancing and fault tolerance guaranteess. The Broker requires to deploy the following instances:

1. A cluster for Node.js with Mongoose, which requires the deployment of two instances. To balance the load, one HAProxy server can be deployed in each instance, and a Virtual IP assigned to one of them. These instances act as balancer of the load of the Broker requests among all Node.js instances. The instances must have a small quantity of available storage, in order to store certificates and configurations.

2. A sharded cluster of MongoDB, which requires the deployment of a mongos[28] (MongoDB Shard) service dedicated for each Node.js instance, and several replica sets, one acting as config server and two acting as shards (to obtain horizontal scalability), each of them composed by three members (to obtain fault tolerance and high availability). The shard instances must be provisioned with enough storage to manage the data related to active requests.

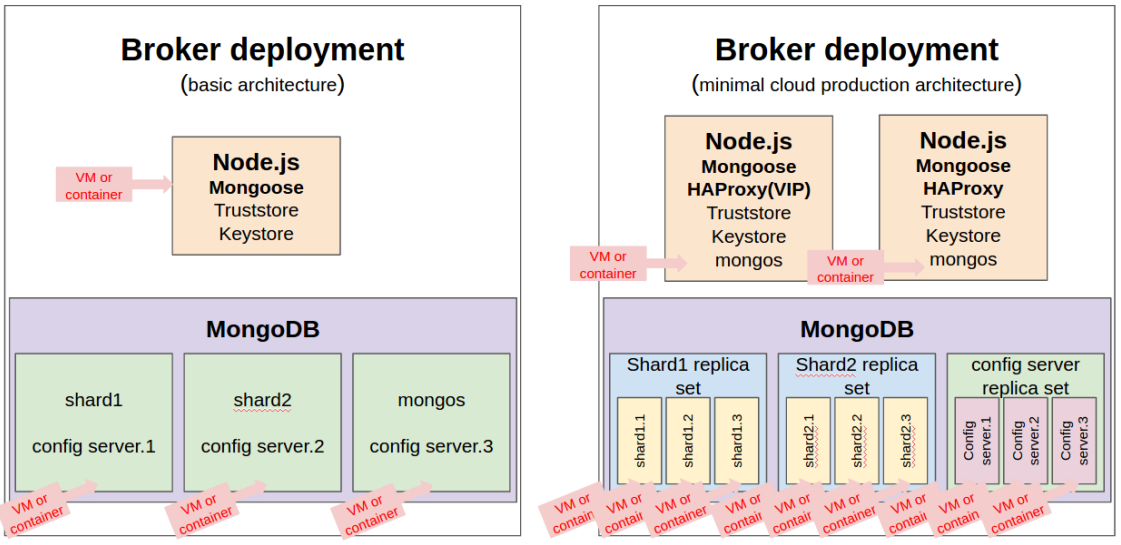

Figure 6: Broker basic vs minimal cloud deployment.

Figure 7 shows the key elements of the Broker architecture that are required in order to achieve scalability in the cloud. To perform upscaling with Node.js and Mongoose it is necessary to deploy another instance and change the load balancer configuration in order to add the new member of the cluster. In the case of MongoDB upscaling is achieved by deploying a new shard replica set with three members and adding its configuration to the config servers. If the operation is downscaling, the process is the opposite. To decide which of the components needs to be upscaled/downscaled it is necessary to monitor the time processing for each request, number of sessions, memory, cpu, load average and disk usage, among other metrics. The scaling process can be triggered automatically with a proper configuration of the Cloudify's monitoring features, defining specific rules in the deployment blueprint.

In each WITDOM domain, a Certification Authority (CA) is needed to issue valid keys and certificates that enable a secure communication between the Broker and each component or service in the corresponding domain. The certificates must be stored into the keystore of the REST-API components that communicate with the broker, in order to guarantee a proper encryption and mutual authentication. This configuration includes mainly the storage of the issued certificate in the Keystore and the domain's CA in the Truststore of each component/service/broker.

[28]Rooting service that processes queries from application layer and determines the location of the data in the sharded cluster.

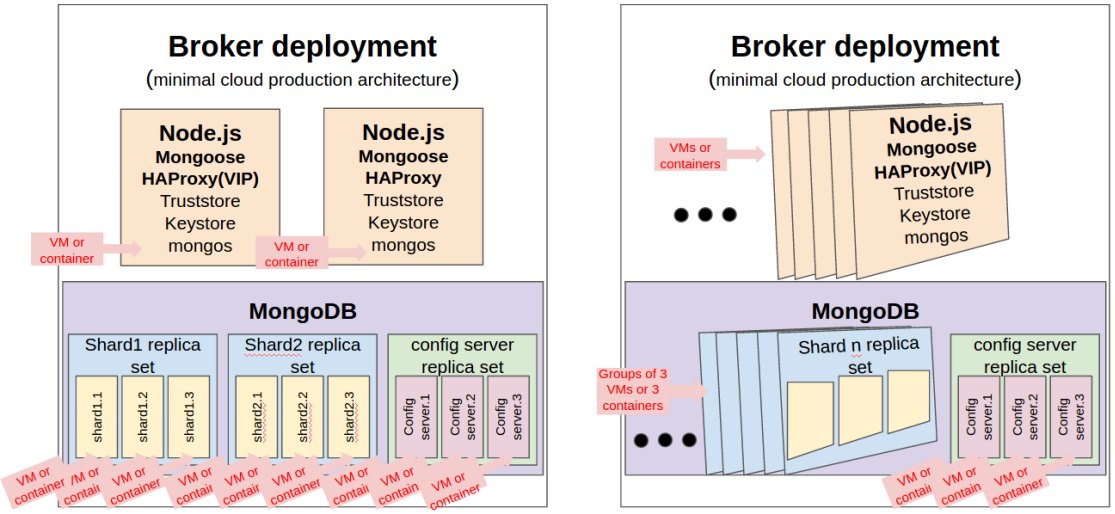

Figure 7: Broker minimal vs scalable cloud deployment.

The trusted and untrusted instances of the Broker are prepared in order to be used as if they were deployed in symmetrically untrusted domains. However, each of them has by definition different keys and certificates issued by different CAs. To enable the mutual authentication between both instances it is necessary to store the untrusted domain's CA certificate (symmetrically) in the Truststore of the trusted domain's Broker. This certificate is used to validate the untrusted domain Broker certificate during the mutual authentication process.

Auditing In addition to its main functionalities, the Broker also implements an auditing module to log the information of all the requests it receives. This module makes use of the open-source library audit-log[29]. This library provides different transport systems for storing the logged information, the system choosen to store the information logged by the auditing module is the mongoose transport. For this reason each Broker instance needs a dedicated MongoDB instance to store the auditing information, which can be adapted to cloud environments in the same way it is done for the MongoDB instance used to store the Broker internal data. All calls received by the Broker are logged and the stored information is the following:

- X-Auth-Token: the user token received in the call.
- The subject of the received client certificate.
- The method of the request (POST|GET).
- The request URI.
- The request headers.
- The request body.

3.3.4 Protection orchestration

The selection of jBPM [30] as the core technology on top of which the Protection Orchestrator has been built, took into account jBPM's cloud capabilities in order to minimize required cloud-specific adaptations. The orchestrator operation basically consists of tracking protection processes states, receiving events (e.g., protection component

[29] https://www.npmjs.com/package/audit-log
[30] http://www.jbpm.org/

finished) and triggering transitions depending on the protection configurations processes specifications. This kind of operation cannot benefit from the main features of the big data infrastructure (which allows to parallelize operations in large amounts of data). Instead, in conjunction with the broker component, the protection orchestrator can be deployed in a cloud-mode, allowing multiple PO instances to respond to a massive number of requests that could not be handled by one unique server (using the broker as load balancer). This deployment mode requires minimal (if not none) modifications in the component. However the following changes (depicted in Figure 8) in the configuration of the deployment would be required:

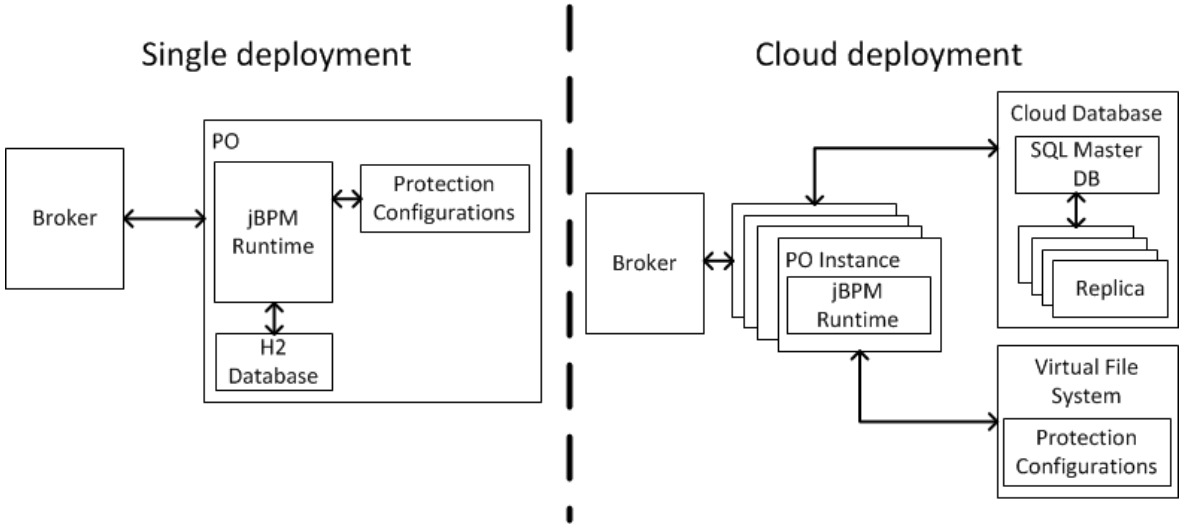

Figure 8: Protection orchestrator single deployment vs cloud deployment.

1. Database: while the PO contains an embedded instance of a H2[31] database in order to manage the persistence of the processes and related data, the PO can use any standard SQL driver to access more scalable databases. By connecting the PO instances to a shared and replicated database, all the POs will automatically share processes states and data. This modification only requires updating a configuration file to properly define the new datasource.

2. Files: The PO relies in the protection configurations, which in the single deployment mode, are stored in the same machine as the PO itself. By using a Virtual File System, which can be shared among all PO instances, all of them will consistently access the same versions of the protection configurations. The location of these protection configurations is also parametrized in a configuration file and should be modified for the cloud deployment mode.

Additionally the cloud deplopyment mode for the jBPM engine itself requires the deployment and usage of Apache Helix[32] and Apache Zookeper[33] to manage the PO instances available at any time. Furthermore, another could-mode approach could be using Docker technology in a cluster way, taking advantages of this system of software containers. In this scenario, multiple Docker containers (having one PO instance each) would be deployed at the same time and a cluster manager like, for example, Kubernetes would be in charge of the system escalation and the management of the containers lifecycle (still using the broker for the load balancing). However, and given the requirements of WITDOM scenarios the number of expected requests to the PO will be easily handled by one unique instance.

[31] http://www.h2database.com/
[32] http://helix.apache.org/
[33] https://zookeeper.apache.org/

3.3.5 Transformer

The stateless design of WITDOM's transformers allows a maximum use of the benefits of cloud deployments (e.g., scalability). Multiple instances of transformers would be able to manage a big amount of transformer requests. However, the scenario and use-cases' characteristics where small number of requests will ask for the transformation of large datasets prevent this approach from truly benefiting from cloud scalability.

For the WITDOM's e-Health prototype, the transformer implements only local parallelization, by making use of the available threads on the host machine. As an alternative approach that can allow to leverage the cloud benefits in this kind of scenario is to be able to parallelize a unique request so one request can be fulfilled by a cluster of worker nodes, instead of one unique server. For this, the transformer service could connect to the big data infrastructure described in Section 3.3.7 in order to parallelize individual transformation requests.

Transformers can have a wide range of responsibilities, from just making format conversions (e.g., FASTQ to WITDOM format) to aggregating, cleaning and or making input data homogeneous (e.g., removing non-valid values in the fraud detection use-case). Whenever a transformer can be adapted to leverage the big data infrastructure, after receiving a transformation request it will connect to the big data infrastructure, instructing it to access the data source, to split the data in chunks of adequate size, and to send these chunks to individual nodes to perform the transformation operation at chunk level. In that case, the big data infrastructure would be responsible to combine all the partial transformations and to store them in the location originally requested by the transformer. The specific transformation logic to be executed by the individual nodes should be informed to the big data infrastructure by the transformer.

3.3.6 Storage

In the context of WITDOM's framework there are two types of storage solutions that are related to the cloud: SQL Storage and Object Storage.

SQL storage The storage component that stores data to be protected, to be outsourced and to be processed (corresponding to components storage and secured storage as identified in the generic architecture in Deliverable D4.2) is implemented by COTS SQL database management systems (DBMS) such as MySQL or MariaDB. Both mentioned DBMS, and most of modern ones, have been designed and implemented with scalability as a core requirement and include deployment modes that directly enables them to benefit from cloud-scalability features, e.g., MariaDB MaxScale[34], MariaDB Galera [35] or MySQL Cluster CGE [36]. WITDOM framework design can also transparently rely on Cloud SQL services hosted by the public cloud provider (e.g, Amazon RDS[37]), instead of hosting the SQL storage software in the public cloud infrastructure.

Object storage WITDOM's ICV protection component (we refer to Deliverable D4.2 for detailed information) offers integrity and consistency verification for data stored in an object storage hosted in the untrusted domain. Similarly, WITDOM's E2EE protection component uses object storage in the untrusted domain to store locally encrypted data. Hence, object storage solutions should also be considered. Object storage approach abstracts some of the lower layers (e.g., files or blocks) of storage away from the administrators and applications. This design principle maximizes the scalability features of object storage solutions and has lead to a majority of cloud storage available in the market to leverage the object storage architecture (e.g., Amazon S3[38] or OpenStack Swift[39]).

[34] https://mariadb.com/kb/en/mariadb-enterprise/mariadb-maxscale/about-maxscale/
[35] https://mariadb.com/kb/en/mariadb/about-galera-replication/
[36] https://www.mysql.com/products/cluster/
[37] https://aws.amazon.com/rds/
[38] https://aws.amazon.com/s3
[39] http://docs.openstack.org/developer/swift/

Both of the data storage approaches for WITDOM architecture described, can be considered cloud-friendly and do not require a cloud-specific layer, it suffices to provide some parametrization during the deployment in order to leverage the benefits of the cloud.

3.3.7 Big data infrastructure

Computing and operating on huge volumes of information is a real challenge due to the emerge of wide-connected devices and new systems or technologies that use different types of data from different kind of sources. Although the information is normally easily available, the problem in fact resides in processing all these variables efficiently, taking into consideration the scalability, robustness and time restrictions of diverse applications. In WITDOM, both scenarios (e-Health and financial) have to deal with these issues due to the huge size of the data that is processed by the applications. In order to simplify the management of these data and improve the efficiency of certain operations, a big data infrastructure is proposed as an extension of the platform.

Different distributed computing frameworks (like Storm, Spark, Flink, etc.), have been developed in recent years in order to try to properly address the previous limitations. In particular, Apache Spark[40] is a big data processing framework built around speed and advanced analytics. It was originally developed in 2009 in UC Berkeleys AMPLab, and open-sourced in 2010 as an Apache project. Spark provides a comprehensive, unified framework to manage big data processing requirements (both batch and real-time streaming) with a variety of data sets that are diverse in nature (text data, numeric data, graph data, etc.). Figure 9 illustrates the big data architecture based on Spark that is used within WITDOM. In particular, it is used by the Anonymization component but may also be integrated in other WITDOM protection components and services.

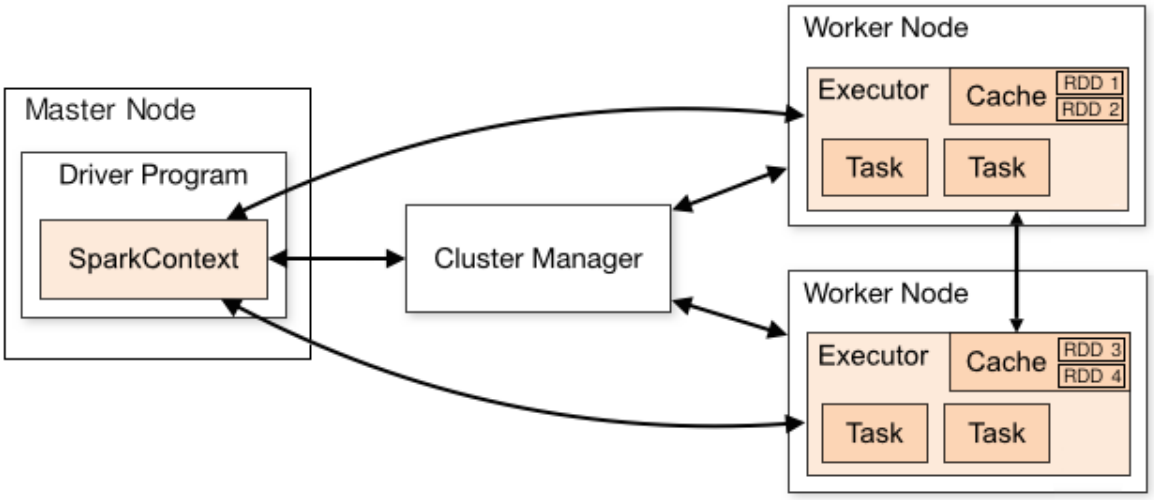

Figure 9: High-level illustration of the big data architecture.

At a high level, every Spark application consists of a driver program that runs the users main function and executes various parallel operations on a cluster of machines maintained in a fault-tolerant architecture. The cluster is generally composed by two types of nodes: a master node for the collection of input data and the execution of the driver or main program, and a number of worker nodes which run distributed application code in the cluster. The main abstraction Spark core provides is a *Resilient Distributed Dataset* or RDD, which is a collection of elements (a software structure containing the input data) distributed or partitioned across all the nodes of the cluster that can be operated on in parallel. Each worker node in the cluster performs a sequence of actions and transformations on these data structures from the instructions programmed in the application code. Although Apache Spark engine is also designed to work on-disk, RDDs commonly persist in memory, allowing it

[40] `https://http://spark.apache.org/`

to be reused efficiently across parallel operations in addition to run programs faster. When RDDs are persisted in memory, each node in the cluster stores any partitions that it computes in memory and reuses them in other actions on that dataset (or datasets derived from it). This allows future actions to be much faster, specially for iterative algorithms or transformations of data. The spark Cluster can connect to several types of cluster managers (either Sparks own standalone cluster manager, Mesos, YARN, OpenStack Sahara, etc.), in order to configure and deploy nodes, in addition to allocate resources across applications. WITDOM has studied the integration of Spark with OpenStack Sahara. Since this integration would be tied to a specific cloud infrastructure it was decided to use a standalone Spark deployment which allows to easily change the underlying cloud platform depending on user requirements.

There are additional components, built on top of the Core API, which are part of the Spark ecosystem and provide additional capabilities in big data analytics and Machine Learning areas. Table 3 shows a brief description of each library.

Component	Description
Spark Streaming	This component on top of Spark's core provides capabilities to perform streaming analytics. It ingests data in mini-batches and performs RDD transformations and actions in real time.
Spark SQL	This module provides support for structured and semi-structured data processing. It defines a programming abstraction called DataFrames and can also act as a distributed SQL query engine.
Spark MLlib	MLlib is Sparks scalable machine learning library consisting of common learning algorithms and utilities, including classification, regression, clustering, collaborative filtering, dimensionality reduction, as well as underlying optimization primitives.
Spark GraphX	GraphX is the Spark API for graphs and graph-parallel computation on top of Apache Spark. this component includes a growing collection of graph algorithms and builders to simplify graph analytics tasks.
BlinkDB	Query engine for running interactive SQL queries on large volumes of data.
Alluxio	A distributed file system enabling reliable file sharing at memory-speed across cluster frameworks with no overhead of disk I/O.

Table 3: List of Spark components.

For distributed storage, Spark can interface with a wide variety of systems, including Hadoop Distributed File System (HDFS), MapR File System (MapR-FS), Cassandra, Elastic, OpenStack Swift, Amazon S3, etc. It also allows for the development of custom interfaces for data collection and storage. In WITDOM, the storage systems used are MySQL and OpenStack Swift, therefore Spark fits perfectly with WITDOM storage requirements.

Spark is written in Scala Programming Language and runs on a Java Virtual Machine (JVM) environment, with a concise and consistent set of APIs (for Scala, Java, and Python programming languages) to create applications using a standard interface.

The big data infrastructure can therefore be deployed as a Spark cluster of VMs or Docker containers, composed by a master node and a set or workers, available for the rest of WITDOM components for on demand distributed processing of both batch and real time streams. For WITDOM we have made available at Docker Hub[41] a Docker image that can work both as master or worker node depending on the input configuration. The deployment of this component enhances the capability of the WITDOM architecture to efficiently process huge

[41] https://hub.docker.com/r/gradiant/spark/

amounts of data with very low latency.

3.4 Protection components

The protection components as introduced in Deliverable D4.2 implement several data protection mechanisms that can be leveraged to protect data that is sent to untrusted environments (public cloud) for storage or processing. The interfaces and internal modules of these components were designed to be flexible and mostly platform-independent, minimizing cloud-specific adaptations.

Deploying the WITDOM platform in a cloud-based infrastructure may require certain security measures. Those are defined using the protection configuration and are delivered to protection components along with a protection request.

One major adaptation of protection components to work in a cloud computing platform is related to packaging and deployment of the components. While in a single end-user environment all WITDOM components may run on a single system (e.g., a computer), however, in a cloud environment, components may run in multiple systems in order to enjoy the benefits of cloud technologies and thus modern deployment systems may be necessary. Cloud deployment systems, such as Cloudify and Docker eases the work of developers and administrators by allowing them to automate the deployment, management, and integration of protection components in the architecture.

In order to benefit from efficient computation on large volumes of data, some protection components also have to be slightly adapted to work with the big data component offered in the adapted WITDOM architecture. The Secure Signal Processing (SSP) Component, for example, makes use of highly parallelizable operations like encryption, decryption, data transformations and homomorphic calculations, which can be parallelized at various levels. Parallelization at the data level may take advantage of the big data infrastructure presented in Section 3.3.7 for distributing the iterative processing of large amounts of data (rows) across several worker nodes.

However, some considerations need to be taken into account when moving a server-side SSP component to the cloud. More specifically, when dealing with secret sharing primitives and interactive protocols across several domains, the timing and data transfers specified by the protocol have to be consistently preserved for it to work properly. Therefore, parallelization below the protocol is not as easy as in the solutions based on a unique domain, as the sequence of operations performed at each interaction round require that all the instances in one domain produce the same answer to a given intermediate request. Hence, the different instances in one domain must be synchronized (e.g., by relying on an instance of a key-pair database, like Redis[42], in their respective domains) to avoid intermediate requests out of the protocol timing. If the server-side SSP component are deployed in a cloud infrastructure where several instances are allowed this synchronization component has to be used.

Another protection component that greatly benefits from the big data infrastructure is the Anonymization component. The deployment of the Anonymization component as a cloud service does not impact significantly on its behavior. The component becomes more scalable, flexible and reliable and, in general terms, it offers a better performance than a pure in-house deployment. However, these features do not differ from the standard behavior of any other cloud service. Note that the Anonymization component operates only in the trusted domain, however, using cloud technology on-premise infrastructure can be used more efficiently on demand, including provisioning and releasing of resources.

The Anonymization component can make use of the big data infrastructure in order to improve the efficiency of its operations. In order to allow this, it includes a big data module that enables the communication with the big data infrastructure (e.g, to request anonymization jobs). This is extremely important for the financial scenario, since it is expected to manage a huge amount of data for each protection operation. In order to enhance the performance of these operations, the Anonymization component can be configured to distribute parts of the processing among different nodes of the big data cluster. This required an adaptation of the anonymization algorithms, so that they could be run over a distributed environment. These adapted algorithms were implemented in the big data module, which is called from the Anonymization component in order to submit a job to the master node of the big data infrastructure.

[42] https://redis.io/

When the Anonymization component receives a request to execute a certain algorithm, its big data module submits this request to the master node of the big data cluster which can connect to the source where data is stored. Once the required information is available, the master node can split up both the data and all tasks for transforming the dataset among the worker nodes in the cluster. All partial results are finally collected and combined in an unique output by the master node, and appropriately stored where necessary.

The anonymization algorithms that can be executed over the big data infrastructure are mainly the generalization, randomization and deletion of data. The first operation implies aggregating the values of some fields, obtaining as a result a new set of limited values. On the other hand, the randomization operation implies adding a certain noise, while the deletion eliminates some of the values from the dataset. All the previous operations are characterized by the fact that they are relatively simple operations that can be easily distributed among several nodes, which fits perfectly with a big data infrastructure. The logic that indicates how the anonymization operation should be applied to the data is given by the given anonymization rules passed to the component. The developed big data module inside the Anonymization component implements the algorithms for generalization and deletion. The randomization algorithm was left out since it is not needed for the fraud detection use-case where the big data infrastructure is demonstrated. However, the effort of developing this algorithm in a distributed fashion should be relatively low, and can be easily implemented later. The big data infrastructure can additionally be leveraged to enable data analyses to determine new sets of anonymization rules.

While in the adapted WITDOM architecture some protection components need to be adapted to the cloud environment, other components such as the End-to-End Encryption (E2EE)[43] and the Integrity and Consistency Verification (ICV) components were developed with focus on secure provisioning of cloud services, and thus apart from adapting them to be deployed with Cloudify and inside containers, such as Docker, no further considerable changes are required.

4 Adoption to scenarios

This section briefly comments on the benefits of transitioning to a cloud deployment for the two scenarios addressed in WITDOM. Due to the large scale of the managed data and the intensive processes carried out on them, both scenarios are good candidates for a cloud implementation, which endows them with better scalability, resiliency and elasticity. There are currently some packages available for the operations in both scenarios, but none of the current proposals address privacy constraints, so their usage is mainly limited to in-house private cloud deployments when privacy is an issue. WITDOM tries to bring about the benefits of using hybrid and public clouds to these scenarios while meeting the privacy constraints, and the cloud adaptation task studies the feasibility of achieving adequate parallelizability and concurrency benefits from a cloud while working on privacy-protected data.

4.1 e-Health scenario in the cloud

The benefits that transitioning to a cloud environment could bring about to the e-Health scenario are unquestionable, and they have been extensively discussed in previous deliverables. In the past, other health-related intensive operations have already benefited from the use of the so called Healthgrids and High Performance Computing mainly for medical imaging and Monte Carlo-based algorithms, but these architectures also present severe privacy issues when outsourcing data outside the security perimeter of the hospital or laboratory.

The e-Health scenario deals with large files (like FASTQ and BAM) and computationally intensive operations, like sequence alignment, which need hours or days to complete. The advent of Next Generation Sequencing techniques is worsening this problem, by increasing the throughput at which new genomic data is produced, therefore increasing the processing needs to interpret and properly analyze these data. Conversely, other simpler operations operating periodically on a large number of small files (e.g., annotation of VCFs) also need a considerable amount of time to finish. These are the two main functionalities (besides backups) that WITDOM tackles in

[43] E2EE was developed in a research project called SPECS `http://www.specs-project.eu/`

the e-Health scenario. Cloud-based computing is therefore likely to improve the speed, quality, and throughput of these operations, enabling more timely and accurate analyses.

4.1.1 Sequence alignment

There are many recent works trying to bring sequence alignment to the cloud, but most of them disregard the privacy problems that this transition opens up. WITDOM, through the proposed architectural design and the developed protection components, seeks to enable the access to some of the aforementioned performance benefits without incurring in the privacy penalties. Some prior recent approaches dealing with sequence alignment in the cloud (mainly over Hadoop), are the following:

CloudBurst [14] is a parallel read-mapping algorithm optimized for mapping next-generation sequence data to the human genome and other reference genomes, for use in a variety of biological analyses including SNP discovery, genotyping, and personal genomics. The used algorithms scale linearly with the number of reads mapped, and with near linear speedup as the number of processors increases.

Vijakumar *et al.* [15] propose an optimized approach to sequence alignment using FASTA algorithm, which incorporates high speed word-to-word matching. The proposed approach relies on Greenplum Massively Parallel Processing (MPP) database and Hadoop/MapReduce. The complex nature of the algorithm, coupled with data and computational parallelism of Hadoop grid and Massively Parallel Processing database for querying from big datasets containing petabytes of sequences, improves the accuracy, speed of sequence alignment and optimizes querying from big datasets.

Kienzler *et al.* [9] devise an incremental data processing model which can hide data transfer latencies while maintaining linear scalability. They implement and evaluate the system for SHRiMP, a well-known software package for NGS read alignment, based on the IBM InfoSphere Streams computing platform deployed on Amazon EC2.

Daugelaite *et al.* [6] introduce the usage of cloud computing for Multiple Sequence Alignment (MSA) of DNA, RNA, and protein sequences, discussing several approaches with a specific focus on the ClustalW and Clustal Omega algorithms. They emphasize the benefits of cloud in these applications.

All the enumerated approaches deal with sequence alignment on Cloud, showing the advantages of parallelizing and elastically growing the used nodes to cope with big files and the computationally intensive alignments; nevertheless, they do not explicitly tackle human DNA and they do not address the privacy problems that outsourcing human genetic sequences entails. Therefore, the idea of using cloud for sequencing was temporarily abandoned for some years; in fact, there are only open-source general-purpose bioinformatics tools like Galaxy[44] or Mercury[45] [13], which can be deployed in a private cloud or in the DNAnexus[46] platform, but none of them are privacy-aware; therefore, the market is currently awaiting for a solution like WITDOM, which enables the power of the cloud while overcoming the privacy constraints that prevent this transition.

4.1.2 Variant annotation

Variant annotation can be a resource demanding application when applied on a large set of files. Several cloud implementations of annotation have been proposed in the recent years, and there are some packages specifically developed for annotation in cloud, besides those integrated in Galaxy or Mercury platforms:

VAT[47] [7] is a variant annotation tool which can be deployed in a private or public cloud, and allows for functionally annotating variants from multiple personal genomes at the transcript level as well as obtaining summary statistics across genes and individuals. It is implemented in C and PHP, and it can also be deployed in Amazon AWS, but it does not provide any kind of privacy protection.

[44] https://galaxyproject.org/
[45] https://www.hgsc.bcm.edu/software/mercury
[46] https://www.dnanexus.com/
[47] http://vat.gersteinlab.org/

CGTag [8] is a genomics toolkit which can be deployed on top of a galaxy cloud instance, and features also annotation. It is programmed in Python, R and Bash. It does not feature any kind of privacy protection.

Actually, SnpEff (See Deliverable D2.2) can be run in a cloud instance with no modifications, either from a command-line interface of through a platform like Galaxy.

As with the alignment case, current solutions for cloud-based deployment of annotation tools do not feature any kind of privacy protection of the outsourced sequences, so their usage is commonly restricted to in-house private clouds, especially in Europe, where the data protection regulation is much more restrictive (see Deliverable D6.3).

4.2 Financial scenario in the cloud

While the e-Health scenario benefits from being deployed in the cloud are bound to domain-specific algorithms (e.g., FASTQ alignment) and data formats (FASTQ files), the financial scenario is more generic and relies on widespread domain-independent statistical and machine learning algorithms (e.g., linear and multi-variable regression, classification algorithms). Hence, the financial scenario can expect the same performance benefits from transitioning these algorithms to the cloud as many other big data analysis scenarios already deployed in the cloud, as they mostly rely in the same kind of algorithms. The main issues that must be addressed in WITDOM regarding the cloudification of the financial-related services whether the same algorithms are applicable and the same benefits remain when instead of dealing with normal data the algorithms work on previously protected data (e.g., anonymized, masked, encrypted). Further details about each of the use-cases are provided in the following paragraphs.

4.2.1 Fraud detection

Fraud detection in credit card transactions in WITDOM follows a machine learning solution based on classification algorithms. Unlike the e-Health scenario, there are many open-source parallelized versions of most common machine learning algorithms that match fraud detection scenario requirements: For example, Apache Mahout [48] provides a wide variety of pre-made machine learning algorithms for different big data platforms (e.g., Spark or Hadoop). For many algorithms not available within Apache Mahout, there are also open-source platform-independent scalable classification algorithms such as parallelizable Support Vector Machines [5]. Finally, and whenever there is no direct matching for the exact requirements and an open-source implementation, the combination of R (chosen end-user interface for WITDOM financial scenario) and Spark (the selected big data infrastructure for WITDOM cloud adaptation layer) SparkR [3] enables non-expert users to parallelize low-level operations (e.g., vector operations or matrix multiplications), which is expected to bring performance improvements to existing algorithms by (almost) transparently leveraging the big data infrastructure.

4.2.2 Risk scoring

The risk scoring use-case, as described in deliverables D2.1 and D2.2, uses machine learning regression algorithms to calculate the parameters that determine the best fit to calculate operations' and customers' risk. As with fraud detection, the algorithms used for risk scoring are well known and there are several open-source implementations that allow to leverage parallelization and, hence, cloud environments under several big data infrastructures:

- Linear regression with R and Hadoop [12] present a way to solve the linear regression model by expressing the least squares solution for the linear regression problem in terms of map-reduce programming paradigm; the latter can be implemented with R and Hadoop using the Rhadoop[49] library.

[48] `http://mahout.apache.org/`

[49] `https://github.com/RevolutionAnalytics/RHadoop`

- Spark already includes a selection of regression algorithms in its machine learning library [2] which could be applicable to the risk scoring use-case: Linear least squares, Lasso, and ridge regression.

As in the fraud detection case, if the aforementioned algorithms are not suitable with the low-level capabilities of R combined with Spark, the end-user in collaboration with WITDOM privacy engineers may try to provide an algorithm that leverages the big data infrastructure.

4.2.3 Cash-flow forecasting

As in the two previous use-cases, the cash-flow forecasting use-case relies on standard statistical methods to implement the required functionalities. For the specific case of forecasting, time series analysis like Autoregressive Integrated Moving Average (ARIMA) are used. Again, given the generality of the algorithm, there are many studies and also open-source implementations with prevalent big data infrastructures that enable to perform such forecasting leveraging the cloud scalability benefits:

- Time series for Spark [4]: a Scala / Java / Python library for interacting with time series data on Apache Spark that provides a set of abstractions for manipulating large time series data sets and models, tests, and functions that enable dealing with these time series from a statistical perspective.
- Altough used for the weather forecasting domain, the same principles and approach taken in [10] could be adapted and applicable for the cash flow forecasting.

5 Implementation guidelines for the cloud scenario

This section extends the implementation guidelines provided in Deliverable D4.2 for the cloud scenario. In particular, these guidelines are gathered through hands-on experience during the implementation of the prototypes in WP5. As discussed in Section 3.2, WITDOM can be deployed with a combination of Cloudify and a configuration management tool. Within the project, in order to simplify the integration process, the developers decided to use the combination of Docker (as further presented in Section 5.1) and Cloudify (detailed in Section 5.2). In practice, other configuration tools like Chef [50], Puppet [51], and Ansible [52] can also be used with WITDOM depending of the preferences of the end-user, however, within the project we use Cloudify which has out of the box integrations with these tools. Guidelines on how to use Chef to deploy WITDOM are provided in Annex D.

5.1 Docker

Developers of WITDOM components should provide a per-component Dockerfile that can be used to rapidly test a given component locally, inside a Docker container, and enable easier integration. *Dockerfile* is a text file containing a *series of instructions* for building a software image that will be run inside a Docker container. It follows a simple syntax - every instruction comprises the *instruction name* (capitalized for readability, for instance, `COPY`) and corresponding *arguments*, separated from the instruction name by a space.

5.1.1 Base image

Instructions in the Dockerfile are executed sequentially. During the image building process, every instruction is run on top of the image resulting from the previous instruction. However, every Dockerfile must begin with an instruction that tells Docker which existing image should be used as a base at the beginning of the image building process for your component. Most likely, the base image will correspond to the target operating system and may

[50] `https://www.chef.io/`
[51] `https://puppet.com/`
[52] `https://www.ansible.com/`

be followed by a tag that specifies its version, for example, `FROM ubuntu:14.04`. The list of existing images is available on DockerHub[53].

5.1.2 Provisioning software

After defining the base image, several `RUN` instructions can be specified that will execute commands provided as arguments. Arguments to the `RUN` instructions are usually shell commands. Several shell commands can be given in the scope of a single `RUN` instruction. Below are some typical examples:

```
RUN apt-get update && apt-get install -y git python-dev python-mysql
RUN pip install virtualenv
RUN git clone https://github.com/project/project.git
RUN cd /home/user/project  && \
      source project/install.sh
```

As evident from the examples above, we use `RUN` instructions to install packages from software repositories, download files, run installation scripts, and navigate through container's filesystem.

5.1.3 Copying files from host

Typically, several files or directories will have to be copied from the host to the Docker container. This usually includes configuration or installation files and a startup script. If we save a file in the same folder as Dockerfile, it can easily be copied with the `COPY` instruction. For instance, `COPY ./start.sh /home/user/` will look for the script `start.sh` in the same directory as the Dockerfile, and copy it to the location on the container's filesystem provided as the second argument. Note that if you change any of the files you copied to the container or their permissions, you will have to rebuild the image for the changes to take effect.

5.1.4 Working directory

An alternative to manually navigating through the container's filesystem from within `RUN` instructions or specifying full paths in the `COPY` instructions is to specify the working directory before a given instruction in the Dockerfile with `WORKDIR`:

```
WORKDIR /home/testuser
RUN mkdir test
```

The `WORKDIR` instruction will create a new directory, if it does not already exist. However, one has to be cautious to always keep track of the working directory from which the next or subsequent instructions will be run.

5.1.5 Exposing ports

When testing deployment of a component inside a Docker container, it is crucial that we expose all of the required communication ports to the host computer. The latter is done with the `EXPOSE` instruction that takes one or more port numbers, for example, if we were running a web server in the container and wanted the host to access it, we would put in the Dockerfile the instruction `EXPOSE 80 443`. However, note that the specified ports will only be exposed when running the container image if an appropriate mapping is configured.

[53] `https://hub.docker.com/`

5.1.6 Running the service

The last Dockerfile instruction has to specify what should be executed when the image will be run inside a Docker container. Typically, we would do this by using the `CMD` instruction and provide a path to the startup script as an argument. This script should start the service provided by a given WITDOM component. Note that the container will not stop running as long as the startup script is executing.

```
WORKDIR /home/user/
CMD ./start.sh
```

The instructions above specify that once the image is run inside the container, a script `start.sh` from the directory `/home/user` on the container will be executed. When copying executables between different types of filesystems, ensure that appropriate execute permissions are set for the startup script prior to the image building process.

Dockerfile supports several other instructions whose descriptions are omitted here for simplicity. A complete Dockerfile reference is available online[54].

5.2 Cloudify blueprints

In order to deploy a component with Cloudify, its logical description (also called a *topology*) must be provided in the form of application *blueprint* in YAML format. Cloudify uses its own domain specific language (DSL) to define blueprints for deploying applications, which is a derivative of the TOSCA standard. Blueprint files should have meaningful, self-explanatory names, matching the name of component whose topology the blueprint contains (for instance, *protection-orchestrator.yaml*).

According to the Cloudify documentation[55], the general structure of a Cloudify blueprint comprises the following parts:

- **Version specification:** Definition of the version of Cloudify's DSL that the blueprint conforms to.
- **Imports:** Lists data type specifications and optional plugins that will be used during the deployment. We usually import Cloudify's types, cloud provider specific types (for instance, OpenStack's, since every cloud provider's infrastructure is slightly different), script plugin, and Docker plugin.
- **Inputs:** Lists input parameters for a blueprint. We use inputs to avoid hard-coding values directly in the blueprint. Instead, input values for all specified input parameters are provided in a separate YAML file (inputs file, for instance *protection-orchestrator-inputs.yaml*).
  For every input parameter that we specify, we can provide a default value directly in the application blueprint. If we do not provide a default value for a specified parameter, a corresponding parameter value must be provided to Cloudify via the inputs file.
- **Node templates:** This is the most important part of the application blueprint. In Cloudify, every application is described as a set of connected nodes, which can be also thought of as resources. Generally we provide *type*, *properties*, *relationships*, and *interfaces* for every node specified in the application blueprint.
  - **Node types:** Nodes can have different types, for instance *web server*, *database*, *application* or *compute type*. These are Cloudify's generic node types. If we specify additional plugins in the imports section of the blueprint, we can use several other node types, for instance, when using the OpenStack plugin, OpenStack-specific types *floating_ip* and *security_group* become available.
  - **Node properties:** Depending on the type of the node, we have to specify several properties that are required by Cloudify in order to correctly deploy a node. The most common properties are for instance *physical (IP) addresses* and *ports*.

[54] `https://docs.docker.com/engine/reference/builder/`
[55] urlhttp://docs.getcloudify.org/3.3.1/blueprints/overview/

  - **Node relationships:** Since the application comprises of several interconnected nodes, we have to tell Cloudify logical relationships among them. Based on a relationship between two nodes, Cloudify can determine the order in which they should be created (for instance, an application node is contained within a virtual machine node, and should therefore be created after the virtual machine node instance).
  - **Node interfaces:** We use interfaces to specify lifecycle operations on a node. For instance, we can tell Cloudify to perform an operation (execute a script) when creating, configuring or deleting the node.

- **Outputs:** The outputs section of the blueprint is usually used to expose application's endpoints in the sense of providing information to the deployer after the deployment has finished. The WITDOM Broker will query deployment outputs to determine where the deployed WITDOM components are located.

The description above outlines the general structure of a blueprint. A single blueprint should be prepared for every WITDOM component, such as, core and protection components, scenario specific components and services. In general, several components could be deployed at once (with a single blueprint), but we decided against it to keep things separated on a logical level.

However, for the components that will be deployed in both trusted and untrusted domain, two versions of the blueprint will have to be provided, since blueprints vary with respect to the cloud provider used when deploying the application. The most notable difference between the two will likely be in the node types used in the blueprints, following the differences in underlying cloud provider infrastructure.

5.3 Docker compose

All WITDOM components are developed and locally tested with the help of the container toolkit Docker. It allows to setup a reproducible development and testing environment. The WITDOM use-cases require different setups with several WITDOM components (including core and protection components), that is, multiple Docker container have to be deployed and connected for each use-case. In order to simplify this process, the Docker Compose tool is used to run and automate the deployment and multi-container Docker applications.

Docker compose is a tool for defining and running multi-container Docker applications. With Docker Compose, we use a YAML file to configure the services for each WITDOM use-case (for each WITDOM application). Then, with a single command, we can create and start all the components / services from the defined configuration.

Using Docker Compose is a three-step process:

1. Define the application's environment with a Dockerfile so it can be reproduced anywhere.
2. Define the components / services that make up the application in `docker-compose.yml` so they can be run together in an isolated environment.
3. Run `docker-compose up` and Compose will start and run the entire application.

An example of a docker-compose YAML file for the WITDOM Backup use-case is provided in Annex C.
Note that Docker Compose has commands for managing the whole lifecycle of an application:

- Start, stop and rebuild services.
- View the status of running services.
- Stream the log output of running services.
- Run a one-off command on a service.

Further details about the functionalities and the usage of the Docker Compose tool can be found online with the official documentation (`https://docs.docker.com/compose/`).

6 Architectural challenges

Hybrid clouds are the prevalent choice in the latest cloud computing trends for IT (71% according to [1]). The most natural explanation for the success of this approach is that it enables organizations to follow their own policies, finding their own balance or trade-offs between public and private clouds and avoiding all-or-none decisions. While the public cloud brings a cost reduction and a great amount of flexibility, a private cloud increases the control over the underlying infrastructure, which provides a higher sense of trust and security.

Whatever the benefits of the hybrid clouds are, it also brings its own set of challenges that must be consistently solved in order to successfully integrate public and private clouds. Not addressing these challenges will inevitably lead to having two separate solutions which will prevent organizations from truly leveraging hybrid cloud benefits.

Within WITDOM most of these challenges were addressed while designing WITDOM's generic architecture, particularly with focus on the two scenarios. Resulting in relatively generic solutions or approaches, they may be leveraged not only on the aforementioned scenarios but also in new contexts and domains.

In what follows, a list of challenges related to hybrid clouds are presented. This includes a short description of each challenge along with measures or decisions that have been considered in the project in an attempt to address them. It must be noted that given the focus of WITDOM on selected technologies, some of these challenges are not directly tackled. As an example, WITDOM is not aiming to improve access control for the cloud.

6.1 Network

Public clouds involved in scenarios where large amounts of data must be periodically pushed to the cloud are constrained by the available bandwidth between the cloud and the data sources.

For the e-Health scenario considered in WITDOM, bandwidth limitations could indeed be an issue. For instance, uploading a FASTQ file with a size of 5GB takes more than 10 hours with a European average Digital Subscriber Line (DSL) or around 30 minutes with an average FTTx line (DSL and FTTx average speeds were extracted from [11]). However, there are no strict requirements regarding the genomic analysis time (results are not expected to be provided before a few days), and according to discussions with the WITDOM e-Health care partner, the bandwidth restrictions will not be considered as a real constraint.

For the financial scenario, some of the considered datasets are much larger than the e-Health ones. Final tests will consider datasets of more than 300GB, which would take more than 24 hours to upload with average EU internet connections. While this could be an issue, the related financial services have been designed to enable partial uploads, that allows frequent uploads of smaller sets of data (e.g., data related to a unique day, week or month). This measure minimizes the constraints once the initial data set is uploaded. It has been discussed with end-users that for this scenario it is reasonable to have to wait more than 24 hours in order to upload the initial dataset. Cloud industry is already dealing with similar situations and most providers include "Import/Export services" which enables to import and export data by using physical hard drives physically shipped to organizations and cloud providers' premises. This option could be provided to WITDOM end-users to accelerate the setup of the framework.

Latency within a cloud environment is another issue. It can be minimized with expensive dedicated connections, but it is physically impossible to completely remove the latency between both private and public cloud infrastructures. After discussions with BBVA, our end-user in the financial scenario, it was concluded that this physical constraint prevents the credit card fraud scoring evaluation use-case to be deployed as a secured service, since it consists on real-time evaluations of credit card transactions to deem whether or not a they are fraudulent.

Finally, given the state of the art approach followed for WITDOM's architecture, most of the network topology, routing and firewall issues are transparent to the framework. However, to address some of the typical issues of cloud infrastructures (e.g., knowing the address where a specific service is deployed, or providing some sort of centralized security and load balancing), WITDOM includes the broker component, which is responsible, for example, for forwarding requests to components, balancing loads between multiple instances of components, and retrying requests whenever the components are non-reachable.

6.2 Portability

Cloud portability refers to the ability of moving applications and data from one cloud environment to another with minimal disruption. This prevents vendor lock-in and therefore also allows the usage of multiple cloud providers at the same time.

The WITDOM use-case scenarios are related to portability. In particular, both, the financial and e-Health use-case involve outsourcing of applications and data from a private (trusted) to a public (untrusted) cloud infrastructure, which is the main objective of WITDOM.

WITDOM framework is designed to minimize the disruption while ensuring that high levels of security and privacy are maintained:

1. **Data** is protected following the protection configuration provided by the developer of secured service in a transparent way for the service's end-user.

2. **Applications** services that are provided in the trusted environment have to be modified so they are capable to work with protected data.

With this approach, end-users have minimum disruption when moving services from the trusted to the untrusted cloud as the process is fully transparent. However, there is indeed some effort to "translate" these services so they can securely operate in the public cloud.

Regarding moving applications and data among public clouds, by choosing to provide all components using a standard container-based technology (Docker in this case), WITDOM avoids the risk of vendor lock-in. During the deployment, WITDOM can leverage compatible vendor-specific container services (e.g., Amazon ECS, Google Container Engine) or by deploying container engines directly in the infrastructure service after starting a virtual machine. Internally, data managed by WITDOM is stored using standard SQL, allowing an easy migration of data from one SQL engine to another.

6.3 Compatibility

In a hybrid cloud approach such as the one promoted within WITDOM there could be some compatibility issues at two different levels.

1. Compatibility of input formats and protection components: in order to avoid having domain-specific protection components, WITDOM framework has agreed a common format for input data that will allow the same PC to protect data from different domains (e.g., a patient's DNAsequence or credit card transactions). In order to keep these transparent to the end-users, WITDOM includes the concept of transformers (to be provided by the secure service developer) that will transform domain-specific formats (e.g., FASTQ files) into a tabular representation that may be stored in SQL databases and processed by protection components. A similar approach has been taken in the opposite direction, transformation services may be invoked as part of the service in order to leave the service outputs in standardized or prevalent domain-specific format.

2. Compatibility of virtual infrastructure: a common challenge for hybrid cloud deployments are the compatibility of the software stacks (e.g., OpenStack vs Microsoft Azure). Again, the decision of using container technologies to deploy the components and services minimizes this challenge as containers should work the same independently from its engine.

6.4 Security

Security is one of the paramount objectives of WITDOM and is a challenge and a key issue whenever there are public clouds involved. There are some mayor aspects that have been introduced in WITDOM framework in order to ensure security:

1. Data protection: every piece of data that is outsourced to the public cloud has undergone a protection process according to the protection configuration.
2. Access control: all calls to end-user services provided within the WITDOM framework are routed through the broker component which ensures there have a valid security token, issued by the WITDOM's IAM, that allows the user to consume such service.
3. Rejection of unknown sources' requests: component will only accept requests from well-known components which will be authenticated my means of certificates. The Broker which will route and authorize most of the interactions among core and protection components will also reject requests from components without adequate certificates.
4. Forbidding communications from the untrusted to the trusted domain: in order to minimize risks, WITDOM has designed the architecture in such a way that no connection from the untrusted to the trusted domain will be required, avoiding the need of, e.g., opening ports in the trusted environment, which may pose some security threats or be against organizations' security policies.
5. Security chain: once the source of the service request has been successfully authorized, all subsequent requests among components are secured by establishing mutual authenticated secure TLS connections between them. I.e. the user calls a service, the broker ensures that there is a token authorizing the user to use such service, the protection orchestrator receives a request from the broker using the mutual authenticated connection, which ensures that the user is implicitly authorized to execute the protection configuration and to outsource that kind of data. When the broker in the untrusted domain receives a request from the trusted domain using the secure connection, it is also certain that it is a valid request and authorizes the execution of the service which will involve accessing protected data. As seen, the core of the security is based on the trust with the previous component of the request chain, this chain goes back to the end-user-broker connection which is established with the aid of the IAM, that acts as the trust anchor.

6.5 Access control

When talking about access control in cloud environments there are two levels to be considered. The first one relates to the control to the infrastructure or the service management while the second one relates to the services provided through the cloud. In a hybrid cloud setting, there is an additional layer as it must be distinguished between the access control in the public and in the private cloud.

WITDOM architecture only addresses the access control to the services themselves, and considers the identity management for the cloud environment itself as out of scope. For service access control, an IAM in the trusted environment is considered, which would actually match the normal setting in scenarios such as the ones considered by WITDOM (financial and e-Health scenarios). This IAM generates PKI tokens which can be checked offline by considering periodically updated lists of revoked tokens. The access to the data in the trusted environment and fine-grained authorizations within the services are out of the scope of the framework itself (even it can be supported with the IAM proposal). For the provision of the financial and e-Health services in any case, best practices should and will be followed. For the access to the data in the end-user or application domain, WITDOM platform will rely in existing access control (e.g., the user must provide credential to access data in a database or upload a file with data, which grants that the user has access to these data). The proposed solution tackles coarse-grained access control (e.g., decides whether if a user can access one service or not). More fine-grained access controls can be included at the end-user application level or in the service itself and also falls out of the scope of WITDOM's framework.

6.6 Management and administration issues

In order to run an application, some parts of the underlying framework may need prior attention from a system administrator, for example, to determine scenario-specific configurations or to define access policies. Below we

give a brief preliminary description of how some of these issues could be addressed in WITDOM. The final approach will be presented in Deliverable D4.6, the final specification of the adaptation layer for cloud computing.

Identity and Access Management The IAM component in WITDOM has to ensure that access privileges are granted according to the access policy and that all users and services are properly authenticated, authorized, and audited. Therefore, before the IAM can be effectively used, an access policy has to be defined. This can either be done by hand by the application administrator or automatically by integrating the IAM component with an existing IAM system. In WITDOM, the IAM component will register users and define their privileges by interacting with organization's own IAM system as specified in requirement #66 in Deliverable D2.1 and D2.2.

In some particular cases the IAM component has to process some additional data to properly define access policies. For example, in the WITDOM e-Health scenario users process data associated with patients, which have to, beforehand, provide written consents that specify who can process what part of their personal data for what purposes (e.g., requirement #118 or #120). It is the role of the WITDOM administrator to prepare these consents in a machine readable format, for example, in XML.

The IAM component is also responsible for auditing. However, it is the role of the administrator to configure the component in terms of what to audit and log, and to set a proper retention policy in compliance with requirements #165 and #176.

Key management The KM component in WITDOM stores all users' encryption keys that the protection components require to protect their data. However, no administration attention is required here because keys are generated either by the KM itself or by the PCs, and access to them is automatically managed by the KM through the use of tokens (for details please see description of the KM component in Deliverable D4.2). A manual revocation of encryption keys is not required, because also in this case, the access to secrets is specified in the scope of the tokens.

Protection orchestrator The PO component in WITDOM stores the protection configurations that describe the agreed data protection steps that must be fulfilled before and after the outsourcing of data. As these protection configurations are entangled with the secured services for which they are developed, the deployment process of these protection configuration should be also related to the deployment of such services. The PO includes a protection configuration management service protected by access-control features that will prevent the usage of such service by users without a WITDOM framework management role. This management service enables to integrate the protection configuration deployment aspects within the secured service deployment.

7 Conclusion

In this deliverable, we have presented the final specification of an adaptation layer for the generic WITDOM architecture proposed in Deliverable D4.2 to cloud computing environments. This deliverable completes the preliminary specification as presented in Deliverable D4.5. In particular, the adaptation layer has been updated and finalized taking into account the recent developments of the prototypes in WP5. We have explored different cloud services and deployment models considering the design of the generic architecture and the requirements of the WITDOM scenarios. In particular, this deliverable described how the components of the generic architecture need to be adapted in order to work in a hybrid cloud model based on OpenStack and Amazon cloud services, and how they can be provisioned, managed and deployed in a way that eases the work of developers in the sense of automation and integration.

We have proposed a deployment system for WITDOM based on Cloudify that fulfills the technology requirements stemming from the chosen cloud computing platforms. The cloud adaptation layer specification proposes specific open-source technologies to implement certain core components such as identity and access management or key management. We have also introduced the big data component for WITDOM that extends the generic architecture and enables efficient processing large scale of data, which is extremely important in e-Health and

financial scenarios. Furthermore, we have also discussed the benefits that transitioning to cloud environments could bring to the WITDOM scenarios.

We have extended our implementation guidelines presented in Deliverable D4.2 with additional deployment methodologies specific for cloud infrastructures. In particular, the presented guidelines following the philosophy as practices in WP5. Finally, we have discussed architectural challenges (e.g., cloud portability) stemming from the adaptation of WITDOM platform for cloud environments.

References

[1] Cloud computing trends: 2016 state of the cloud survey. http://www.rightscale.com/blog/cloud-industry-insights/cloud-computing-trends-2016-state-cloud-survey.

[2] Linear methods - spark.mllib. https://spark.apache.org/docs/1.6.2/mllib-linear-methods.html.

[3] Sparkr (r on spark). https://spark.apache.org/docs/latest/sparkr.html.

[4] Time series for spark (the spark-ts package). https://github.com/sryza/spark-timeseries.

[5] Edward Y. Chang, Kaihua Zhu, Hao Wang, Hongjie Bai, Jian Li, Zhihuan Qiu, and Hang Cui. Psvm: Parallelizing support vector machines on distributed computers. In *NIPS*, 2007. Software available at `http://openbigdatagroup.github.io/psvm`.

[6] Jurate Daugelaite, Aisling O' Driscoll, and Roy D. Sleator. An overview of multiple sequence alignments and cloud computing in bioinformatics. *ISRN Biomathematics*, 2013.

[7] Lukas Habegger, Suganthi Balasubramanian, David Z. Chen, Ekta Khurana, Andrea Sboner, Arif Harmanci, Joel Rozowsky, Declan Clarke, Michael Snyder, and Mark Gerstein. Vat: a computational framework to functionally annotate variants in personal genomes within a cloud-computing environment. *Bioinformatics*, 28(17):2267–2269, 2012.

[8] S. Hiltemann, H. Mei, M. de Hollander, I. Palli, P. van der Spek, G. Jenster, and A. Stubbs. Cgtag: complete genomics toolkit and annotation in a cloud-based galaxy. *GigaScience*, 3(1), 2014.

[9] Romeo Kienzler, Rémy Bruggmann, Anand Ranganathan, and Nesime Tatbul. *Large-Scale DNA Sequence Analysis in the Cloud: A Stream-Based Approach*, pages 467–476. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[10] L Li, Z Ma, L Liu, and Y Fan. Hadoop-based arima algorithm and its application in weather forecast. *International Journal of Database Theory & Application*, (6):2013.

[11] SamKnows Limited. Quality of broadband services in the eu. Technical report, October 2014.

[12] Bogdan OANCEA. Linear regression with r and hadoop. `http://cks.univnt.ro/uploads/cks_2015_articles/index.php?dir=12_IT_in_social_sciences%2F&download=CKS+2015_IT_in_social_sciences_art.144.pdf`.

[13] Jeffrey G. Reid, Andrew Carroll, Narayanan Veeraraghavan, Mahmoud Dahdouli, Andreas Sundquist, Adam English, Matthew Bainbridge, Simon White, William Salerno, Christian Buhay, Fuli Yu, Donna Muzny, Richard Daly, Geoff Duyk, Richard A. Gibbs, and Eric Boerwinkle. Launching genomics into the cloud: deployment of mercury, a next generation sequence analysis pipeline. *BMC Bioinformatics*, 15(1):1–11, 2014.

[14] Michael C. Schatz. Cloudburst: highly sensitive read mapping with mapreduce. *Bioinformatics*, 25(11):1363–1369, 2009.

[15] S. Vijayakumar, A. Bhargavi, U. Praseeda, and S. A. Ahamed. Optimizing sequence alignment in cloud using hadoop and mpp database. In *2012 IEEE 5th International Conference on Cloud Computing (CLOUD)*, pages 819–827, June 2012.

Appendices

A Cloudify blueprint for Barbican

```
1. tosca_definitions_version: cloudify_dsl_1_2
2.
3. imports:
4.   - http://www.getcloudify.org/spec/cloudify/3.3/types.yaml
5.   - http://www.getcloudify.org/spec/fabric-plugin/1.3.1/plugin.yaml
6.   - http://www.getcloudify.org/spec/openstack-plugin/1.3.1/plugin.yaml
7.   - http://getcloudify.org/spec/chef-plugin/1.3.1/plugin.yaml
8.
9.  # Declaration of required inputs
10. # (to be read from another file at the time of deployment)
11. inputs:
12.   # Barbican VM settings
13.   vm_barbican_image:
14.     description: Image used when launching VM with Barbican service
15.   vm_barbican_flavor:
16.     description: Flavor of the VM that will be running Barbican service
17.   vm_barbican_user:
18.     description: User for connecting to agent VM's
19.   vm_barbican_ip:
20.     description: Floating IP
21.
22.   # Chef settings
23.   chef_server:
24.     description: URL where the Chef-server is accessible
25.   chef_version:
26.     description: Version of Chef we are using
27.   chef_client_name:
28.     description: Chef Server validation client name
29.   chef_validation_key:
30.     description: Chef Server validation key
31.   chef_node_prefix:
32.     description: Chef node prefix (as seen on Chef-server)
33.   chef_node_suffix:
34.     description: Chef node suffix (as seen on Chef-server)
35.
36. node_templates:
37.   # VM Server node For Barbican
38.   vm_barbican:
39.     type: cloudify.openstack.nodes.Server
40.     properties:
41.       agent_config:
42.         install_method: remote
43.         user: { get_input: vm_barbican_user }
44.       server:
45.         image_name: { get_input: vm_barbican_image }
46.         flavor_name: { get_input: vm_barbican_flavor }
47.
48.     relationships:
49.       - target: barbican_floating_ip
50.         type: cloudify.openstack.server_connected_to_floating_ip
51.       - target: barbican_security_group
52.         type: cloudify.openstack.server_connected_to_security_group
53.
54.   barbican_floating_ip:
55.     type: cloudify.openstack.nodes.FloatingIP
```

```
56.     properties:
57.       use_external_resource: true
58.       resource_id: { get_input: vm_barbican_ip }
59.
60.   barbican_security_group:
61.     type: cloudify.openstack.nodes.SecurityGroup
62.     properties:
63.        use_external_resource: true
64.        resource_id: sg-idm # Existing OpenStack security group
65.
66.   chef_node_barbican:
67.     type: cloudify.chef.nodes.ApplicationServer
68.     properties:
69.       chef_config:
70.         version: { get_input: chef_version }
71.         chef_server_url:  { get_input: chef_server }
72.         environment: _default
73.         node_name_prefix: { get_input: chef_node_prefix }
74.         node_name_suffix: { get_input: chef_node_suffix }
75.         validation_client_name: { get_input: chef_client_name }
76.         validation_key: { get_input: chef_validation_key }
77.         runlists:
78.          configure: 'recipe[barbican]'
79.         attributes:
80.           floatingip: { get_input: vm_barbican_ip }
81.
82. # Output info about Barbican service, perhaps required by Broker
83. outputs:
84.   barbican_info:
85.     description: Barbican service info
86.     value:
87.       barbican_internal_ip:
88.          { get_attribute: [ vm_barbican, ip ] }
89.       barbican_floating_ip:
90.          { get_attribute: [ barbican_floating_ip, floating_ip_address ] }
91.       barbican_vm_name:
92.          { get_attribute: [ vm_barbican, external_name ] }
93.       barbican_endpoint:
94.          { concat:
95.             ['https://',
96.              { get_attribute: [barbican_floating_ip, floating_ip_address] },
97.               ':9311' ] }
```

B OpenStack Barbican and Keystone Dockerfile

```
# The base image from which we are building
FROM ubuntu:14.04
LABEL Description="This image starts OpenStack services Keystone and Barbican and
configures Barbican to use Keystone token for authentication."

RUN apt-get update && apt-get install -y \
        build-essential \
        git \
        keystone \
        libffi-dev \
        libldap2-dev \
        libsasl2-dev \
        libssl-dev \
        libsqlite3-dev \
        libxml2-dev \
        libxslt1-dev \
        mysql-server-5.5 \
        python-dev \
        python-mysqldb \
        python-pip

# Install Barbican and it's dependencies
RUN pip install \
                alembic \
                lxml \
                pecan \
                python-ldap \
                pytz \
                uwsgi \
                virtualenv \
                virtualenvwrapper
                #\
                #/home/user/barbican

RUN pip install --upgrade pbr

RUN pip install \
                python-keystoneclient==3.1.0 \
                python-openstackclient \
                wrapt

###################################
# KEYSTONE SETUP AND CONFIGURATION
###################################
# Copy Keystone configuration file to container
COPY ./keystone.conf /etc/keystone/

# Copy script that will be used when initializing keystone to container
COPY ./initialize_data.py /root/

###################################
# BARBICAN SETUP AND CONFIGURATION
###################################
# Clone Barbican repository from Git
RUN git clone https://github.com/openstack/barbican.git
RUN cd barbican && \
        virtualenv .barbicanenv  && \
        . .barbicanenv/bin/activate && \
```

```
        pip install uwsgi && \
        pip install -e \$PWD
        #cd /root/barbican/ \
        #pip install /root/barbican/
        #./bin/barbican.sh install

RUN  mkdir /etc/barbican && \
         mkdir /var/lib/barbican &&\
         chown $(whoami) /etc/barbican && \
         chown $(whoami) /var/lib/barbican && \
         cd barbican && \
         cp -r etc/barbican/ /etc/

# Copy Barbican configuration to apropriate location
COPY ./barbican-api-paste.ini /etc/barbican/barbican-api-paste.ini


# Copy startup script that initializes and starts both services
COPY ./start.sh /

CMD ./start.sh

# Expose Barbican and Keystone endpoints
# Keystone public, Keystone admin, Barbican public
EXPOSE 5000 35357 9311
#9312 is Barbican admin
```

C Docker-compose YAML for the Backup use-case

```
# FOR COMPATIBILITY PURPOSES; CHANGE ASAP
networks:
  default:
    external:
      name: brokerobjectstorage_default

#######################################################
# Configures and deploys WITDOM protection components
# and auxiliary components needed for the E-Health
# backup use-case
#######################################################
services:

  # End-to-End Encryption client (trusted)
  e2ee-client:
    container_name: e2ee-client
    build: ../../components/pc/e2ee/client
    ports:
      - 1112:8080
    restart: on-failure
    depends_on:
      - e2ee-server

  # End-to-End Encryption server (untrusted)
  e2ee-server:
    container_name: e2ee-server
    build: ../../components/pc/e2ee/server/docker
    ports:
      - 1111:4443
    depends_on:
      - postgres

  # Postgres database for E2EE-server
  postgres:
    image: sameersbn/postgresql:9.5-3
    container_name: postgres
    restart: always
    environment:
      - PG_PASSWORD=passwd
      - DB_NAME=e2ee
    ports:
      - 5432:5432

  # Integrity and Consistency verification client (trusted)
  icv:
    container_name: icv
    build: ../../components/pc/icv/client
    ports:
      - 8090:8090
    depends_on:
      - icv-server

  # Integrity and Consistency verification server - VICOS server (untrusted)
  icv-server:
    container_name: icv-server
    build: ../../components/pc/icv/server/docker # CHECK WHAT IS ON CD BRANCH
    ports:
      - 2775:2775
```

```
    environment:
      - SERVER_IP=icv-server
    tty: true
```

D WITDOM deployment using Chef

Once a component has been developed and tested locally, component developers should provide a Chef repository that is used to automate the deployment of the component in the cloud. Although we usually talk about Chef recipes when automating component setup, a recipe is usually accompanied by other files. One or more recipes are contained within a cookbook, which is a part of a broader concept, a Chef repository.

D.1 Chef variants

Chef has several flavors, including *chef-solo* and *chef-server*. Chef-server allows centralized storage of Chef cookbooks, which are distributed to Chef-clients. Chef-server will be used when deploying components to the cloud. However, the easiest way for developers to test whether their WITDOM component?s software provisioning process runs as expected, is by using the Chef-solo, which does not require setup of Chef-server. Chef Development Kit[56] must also be installed, which packages all the tools necessary for the development process, including *Knife*, a tool for managing Chef repositories, and *Berkshelf*, a cookbook dependency manager.

D.2 Chef repository

Chef repository is a directory that holds Chef cookbooks, roles, environments, data bags and more. During the development and testing stage, Chef repository is usually located on the workstation. However, once all the automation scripts have been tested locally, and are prepared for deployment to the cloud, repositories will be uploaded to a Chef server in the cloud. Chef clients will be installed on newly provisioned cloud VMs with the help of Cloudify's Chef plugin and pull the appropriate repositories from Chef server.

Chef repository comprises of several subdirectories. Here we focus on two of them, which we believe are the most important: *cookbooks* and *roles*.

Cookbooks A cookbook represents a unit of configuration. It defines a software automation scenario and contains all the files that are required to support it. Cookbooks should have self-explanatory names, after the software whose installation and configuration they automate. For instance, examples of existing cookbooks are *git*, *redis*, *mysql*, *nodejs*, *nginx*, *apache2*, *php*, and *java*.

There is a variety of existing cookbooks that allow us to re-use code for automatic setup of software. Although component developers will be asked to provide one cookbook for their component, it is likely (but not required) that this cookbook will depend on several existing cookbooks. For instance, it is very common to use cookbooks such as *apt* (for updating software package lists), *git* (for installation of git versioning system client) or cookbooks that provision databases and web servers. Therefore, before starting the development of a component?s own cookbook, it is recommended to browse through Chef Supermarket[57] to see whether existing cookbooks can be used to aid the automation process. Otherwise, developers will likely have to write the code to set up not only the core of their WITDOM component, but all of its dependencies as well.

For every cookbook, Chef requires some *metadata*. Cookbook metadata is placed in a Ruby file *metadata.rb* inside the cookbooks directory. These metadata include cookbook description, license, maintainer information and cookbook dependencies.

A cookbook contains one or more *recipes*. Usually we will also need *templates*, and perhaps *attributes*. All of these are represented as subdirectories of the cookbooks directory.

- **Recipes:** Recipe is a file in programming language Ruby and can be thought of as a collection of resources. Every cookbook should have at least one recipe, the default one (*default.rb*). The default recipe is generated when we create an empty cookbook with tools like Knife. If necessary, Ruby code can be split

[56] `https://supermarket.chef.io`
[57] `https://downloads.chef.io/chef-dk/`

into several recipes. A recipe can optionally depend on one or more other recipes and may even include other recipes from external cookbooks.

Some of the most commonly encountered resources used in Chef recipes include *package*, *file*, *remote_file*, *directory*, *service*, and *execute* resources. Some simple examples are given below:

- Installing software packages:

```
package [?python?, ?python-dev?, ?python-pip?]
```

- Executing shell commands:

```
execute ?Running configure script? do
  command ?./configure.sh?
end
```

- Creating a directory:

```
directory ?/home/user/newDirectory' do
  owner 'user'
  recursive true
end
```

- Managing services:

```
service ?mysql? do
  action :start
end
```

As evident from the examples above, every resource has several configurable properties. For a full list of resources and their properties, WITDOM developers should refer to online documentation[58]. Note that some existing cookbooks provide their own resources that can be used within your own recipes. In this case, it is necessary to specify that your cookbook depends on another cookbook. Cookbook dependencies are managed by tools such as Librarian or Berkshelf.

- **Templates:** Template is a file with Embedded Ruby code (.erb). Typically we use templates for configuration files. We take a configuration file and replace the desired configuration (for instance IP addresses or any other runtime and user-defined configuration) with Ruby variables. Then, we can use the template resource in the recipe to determine where the template should be saved on the target system and specify the values that will substitute embedded Ruby code.

- **Attributes:** Attribute files are sometimes used in cookbooks to provide default values for variables used within Chef recipes. For instance, instead of hard-coding a path in a recipe like this:

```
execute ?Installing component? do
  command ?./home/user/km/src/install.sh?
end
```

We would put this in the default attribute file (*attributes/default.rb*):

```
default[?key-manager?][?dir?] = ?/home/user/km/src?
```

Then in the recipe, we would use the above definition like this:

[58] https://docs.chef.io/resources.html

```
km_home = node[?key-manager?][?dir?]
execute ?Installing component? do
  command ?.#{km-home}/install.sh?
end
```

Attribute files are not mandatory. We can still reference attributes in recipes even if we do not declare them in a file, but in this case, we have to provide their values prior to the Chef run. This can be done with roles.

Roles Roles enable us to package operations that must be executed during the automation process as a single job. They can be written either in JSON or in Ruby. Each role defines a *run-list* and values for attributes (if applicable).

Run-lists are ordered lists of recipes and/or roles that are executed sequentially. When Chef-solo or Chef-client will be run on a target node, it will look for a run-list in order to determine in which order software should be installed. Roles are the preferred way for specifying and WITDOM component developers are encouraged to use them.

Note that the description of a Chef repository provided in this section omits definition of some other concepts that can be used, such as *environments* and *data bags*. We also reduced a complete description of cookbooks to only include what we believe should suffice for automated deployment of components. Although WITDOM component developers are encouraged to use Chef in its full capacity, providing a cookbook and a role with a run-list is the most crucial part from the deployment perspective.

D.3 Steps to guide the development

To summarize, the development process of automation software with Chef will include the following steps:

1. Creating an empty Chef repository.
2. Browsing through the Chef Supermarket in order to find if any of the existing cookbooks (and recipes contained within them) can be used to aid the setup of a given WITDOM component.
3. Creating a new cookbook for a WITDOM component.
4. Providing metadata for newly created cookbook, which may include dependencies on other cookbooks.
5. Writing a recipe or a set of recipes that automate your WITDOM component?s installation and configuration. If dynamic configuration is necessary, templates should be created for configuration files and referenced from a recipe.
6. Creating a role, which should contain a run-list and provide attribute values, if they are used inside recipes.
7. Testing the automated deployment of the component with chef-solo.