A Generic Architecture for Scalable and Highly Available Content Serving Applications in the Cloud

Evie Kassela, Ioannis Konstantinou and Nectarios Koziris
CSLAB, National Technical University of Athens
{evie,ikons,nkoziris}@cslab.ece.ntua.gr

***Abstract*—The cloud computing paradigm allows service providers to offer scalable and highly available applications to their end users. Typical cases where this is required are content serving applications, where a large number of connected users manage arbitrary data amounts. In the Big Data era, where the amount of information that is being produced and consumed grows exponentially, centralized legacy approaches are inefficient, as they cannot adequately scale according to the number of connected users or the dataset sizes. In these cases, an efficient cloudification of content serving applications is required in order to benefit from the cloud's offerings. In this work, we present a generic architecture that can be used by almost any content serving application in order to offer scalable and highly available data management operations to their users by employing cloud management techniques. We describe the architectural blocks of our approach along with how they can be efficiently deployed in a cloud environment. We document our experiences with an actual deployment of a typical content serving application over okeanos, an Openstack compatible public cloud service. We describe the open source frameworks that we have selected from a plethora of existing tools, we justify our choices and we describe our initial observations during their operation. We give a detailed overview of how we installed and configured these systems to achieve high availability and scalability in a public cloud setting. Finally, we document our initial performance evaluation where we showcase the system's ability to handle increasing workloads by elastically scaling its resources.**

I. Introduction

A typical need for many applications is the management and handling of the digital content that they produce, consume or serve. Applications such as e-learning [19], digital libraries [5], news sites and generic enterprise content management [12] (e.g., search, collaboration, records management, digital asset management, workflow management, content and scanning) are typical examples. In that cases, a number of different system categories are employed to solve the encountered challenges, according to the specific use cases [16]. Content, document, information and knowledge management systems are typical candidates to solve the respective needs.

Irrespective the application in hand, these systems follow the same pattern in terms of software design and module deployment: they employ a web serving front-end which implements the respective business logic and acts as a gateway to the users, and a data storage back-end where content metadata is being stored, such as a typical RDBMS. In more complex scenarios, a storage backend such as a file system is being utilized, in order to store "raw data" that is not suitable for the RDBMS, because, for instance, of their size or schema. Finally, when extra processing is required to be done in either the "raw data" or the metadata, a processing engine is utilized to perform this task.

In essence, these systems are offered as bundled web application frameworks [2] which are utilized by software developers to design and implement the required solution. Systems such as Drupal [20], Django-CMS [6] and Jahia are a small subset. These frameworks are based on various tools and can support a variety of implementations, software languages and databases. The typical vanilla setup of such systems follows a centralized approach, where web front-ends, databases and, if any, processing and storage engines utilize a single instance (i.e., a single machine) that serves the entire workload. Not only legacy deployments, but even modern ones are still utilizing this setup, due to its simplicity, ease of installation and maintenance.

Although this setup was sufficient for the majority of the applications up till now, the past few years this has been radically changing. The explosion of the data being created, consumed, stored and processed requires different approaches to deal with the arising issues of the "Data Deluge" [4], or the so called Big Data [13]. Many organizations from different domains worldwide are coming across huge big data needs, and this requires a different approach on the way content management systems are being designed and deployed [3]. Moreover, companies that offer their services to end-users over the Internet (for instance, social networking or Internet gaming sites) typically experience varying resource demands according to user traffic [15].

Taking into account the previous considerations, it is clear that a scalable approach is needed both in the design and the deployment, to handle both increasing and irregular load patterns. As both the volume and the type of required resources vary over time, legacy approaches that consist of static application deployments over private data-centers are not an option anymore. On one hand, the up-front cost of building such infrastructures is totally prohibitive for small companies that cannot tell beforehand whether their business will grow sufficiently enough to pay off this initial cost. On the other hand, even in the case where a private infrastructure is already present, static deployments lead to sub-optimal resource usage, due to time-varying resource demand per application or component.

The answer to large up-front infrastructure costs, varying resource needs and static resource allocation problems is cloud computing. The pay-as-you go nature of public cloud providers enables companies to allocate the necessary amount of resources for as long as they want and being charged on a fine-grained manner for the exact resource usage. Even in

978-1-4673-7741-6/15 $31.00 © 2015 IEEE
DOI 10.1109/NCCA.2015.22

the case of private infrastructures where the pay-as-you go feature does not apply, the virtualization capabilities offered by private cloud solutions can easily tackle static resource allocation problems. Therefore, whether inside a private datacenter, or entirely in the cloud, elastic platforms offer the dual advantage of better resource utilization and cost minimization [8], a win-win situation for both SMEs and larger companies.

Nevertheless, cloud technology is not the silver bullet that "automagically" provides these characteristics out of the box for any possible content serving application. Although the cloud infrastructure can easily grow or shrink according to user needs, this is not the case for applications running on top of it. The majority of the web application frameworks [2] cannot easily handle cloud deployments, and if they offer cloud extensions, they are limited to simple automations of centralized deployments, i.e., without offering scalable solutions.

There are commercial "PaaS" offerings such as Google's AppEngine [17], Microsoft's Azure [10], Heroku [14], CloudBees[1], and Engine Yard[2] that can be used to easily create and deploy a scalable cloud application on premises. The resource management is then performed automatically by the provider, without allowing the user to control the underlying utilized resources: users access a sandboxed system in which they have only a limited set of available operations to perform, without the ability to monitor and manage underlying IaaS related aspects, since this is performed by the provider on a user-agnostic manner.

In the case where the application user wants to have complete control over his application, a different approach is required, where the "IaaS" feature of cloud computing is utilized. Proprietary systems like Amazon's Elastic Beanstalk[3] and OpsWorks[4], Redhat's openshift[5], Vmware's cloud foundry[6] and Jelastic[7] are being utilized. Nevertheless, these systems are proprietary and can easily lead to a vendor-lock-in.

To avoid vendor-specific implementations, there are a number of open source configuration management systems [1], such as Ansible, Chef, Puppet, etc. These systems utilize a scripting language to describe complex application setups and offer a mechanism to implement the entire application deployment using the execution of simple self-contained scripts.

Taking into account the previous shortcomings, in this paper we present a generic cloud-enabled architecture for content serving applications that employs scalability, high availability and automated deployment functionalities. More specific, we make the following contributions:

- We document the design and implementation of a scalable architecture for serving, indexing and storing data on a cloud-based infrastructure. We describe the design and deployment of high availability mechanisms that we adopted.
- We describe the automation of the infrastructure installation configuration and management (scalability and recovery) procedure by utilizing configuration management scripts, following the DevOps [7] paradigm. We discuss our approach towards elastic cloud deployment.
- We showcase the architecture's ability to integrate different tools and deploy the infrastructure to any private or public cloud service. We describe how this is achieved with the use of open sourced technology and non-proprietary software. We used open source tools for our implementation and a free cloud vendor.
- We present our initial experimental results where we showcase that our architecture can scale almost linearly and handle increasing workloads by simply adding more infrastructure resources.

The structure of the paper is the following: In Section II we give a brief overview of the basic sub-modules that a cloud-enabled content serving system consists of. In Section III we describe the proposed architecture along with some implementation details and a discussion about extending automations. In Section IV we give a preliminary performance analysis and in Section V we conclude our work.

II. Background

A. Types of data, processing and storage systems

In this section we describe the different data types and their storage requirements as well as the need for data processing and propose the usage of appropriate systems. We also mention typical data management use cases.

When talking about Big Data the first thing that we should focus on is the efficient storage of many large datasets. One dataset, therefore mentioned as resource, consists of two types of data: the raw data (GB/TB scale) and the description (metadata - MB/GB scale). Metadata are typically one order of magnitude smaller than the raw data.

Raw data may have a semi-structured or unstructured form and may consist of one or more data files. Metadata describe the contents and context of the data files. The main purpose of metadata is to facilitate in the discovery of relevant information and they are used for searching. Metadata store and provide information about one or more aspects of the raw data, such as the means of creation of the data, time and date of creation, the creator or author and the standards used.

In case data analytics are also requested on these large raw datasets, a distributed processing engine can be used. Typical data analytics cases are for example pattern recognition, machine learning, etc.

Because of the different storage requirements for metadata and raw data, we employ different storage systems for each of them. We also use a separate distributed processing system for running resource hungry data analytics. We have selected to design these systems with focus in scalability and high availability and therefore we use cloud services for compute and storage requirements. These are the proposed systems, as shown in Figure 1:

a. **Metadata Storage and Serving System:** This storage system contains data in the order of MB or a few GB that are fully structured. These data are inserted in the infrastructure by registered users when they describe a new resource and can be accessed by any user during a search operation (i.e., a read query). The type of these data is write-once read-many, and therefore our storage solution needs to be optimized for

[1] http://www.cloudbees.com/
[2] https://www.engineyard.com/
[3] http://aws.amazon.com/elasticbeanstalk/
[4] http://aws.amazon.com/opsworks/
[5] https://www.openshift.com/
[6] http://www.cloudfoundry.com/
[7] http://jelastic.com/

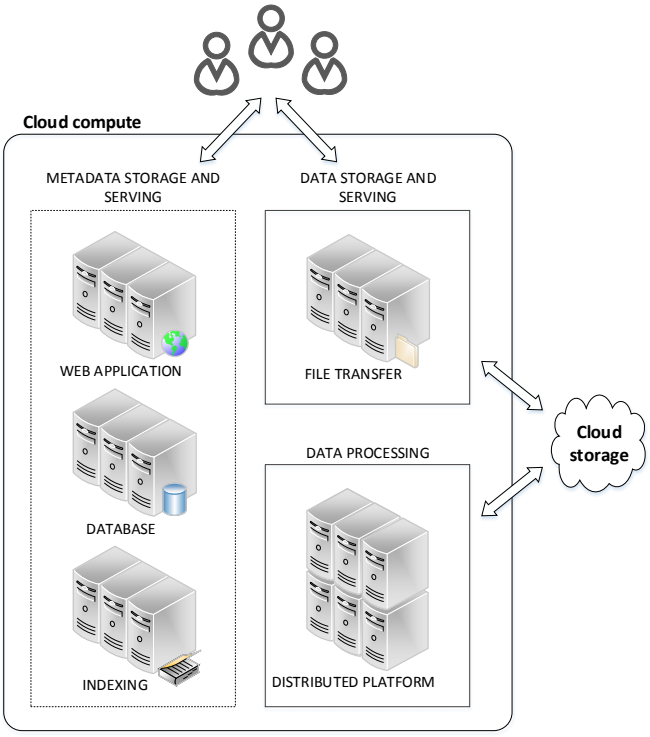

Fig. 1: Cloud-based content management system

heavy reads. Also full-text search is supported by creating lucene type indices.

b. **Data Storage and Serving System:** This storage system contains the resources' raw data that can be in the order of GB per resource and in the order of TB in total. Raw data are served to registered users for download. Because multiple users may request access to many resources the infrastructure needs to have an increased data network bandwidth to serve the load of requests.

c. **Distributed Processing System:** This system takes as input the type of processing and the input and output data locations in the data storage system. A virtual cluster will be in stand-by with the necessary software components in order to process the data and return the results.

B. Data management cases

Apart from application-specific use cases, the same generic functionalities are offered by most content management systems for resource management. In particular, users can usually perform the following data management actions through the web application:

1. *Insert a new resource:* A registered user will be able to describe and upload a resource. The resource will receive a globally unique identifier (GUID), its metadata will be stored in the metadata storage system and the raw data in the data storage system.
2. *Search for a resource:* Any user will be able to make queries in order to find the resource of interest. The search is based on resources' descriptions so these queries will be applied to the indices built for the resources' metadata.
3. *Download a resource:* A registered user will be able to download the raw data of a resource that interests him.
4. *Process a resource:* A registered user will be able to select an available processing tool, define as input a resource and the processing result will be stored as a new annotated resource in the data storage system.

III. System Description

Based on our background analysis we implemented the following subsystems using the undermentioned selected open source tools. The tools were appropriately configured in order to achieve optimal overall system performance.

A. Metadata Storage and Serving System

This subsystem stores the resources' metadata and allows users to search them. It consists of a web front-end and a storage back-end. In both the web front-end and storage back-end layers scalability and high availability are required.

For the web front-end we utilize HAproxy[8], a software load balancer that distributes the requests to a number of webservers in a round-robin fashion. As requirements increase, new webservers can be launched and added to the load balanced cluster. In each webserver the same web application runs as the site frontend. We used the Django [6] web application framework for our implementation.

For the storage back-end a PostgreSQL cluster is used in master-slave mode. This means that one server is allowed to modify the data (master) and the other servers track changes in the master (slaves). Each slave has therefore a full copy of the master's data. All slaves are allowed to serve read-only queries, i.e. they are in hot-standby mode, for load balancing purposes. A software load balancer is also used here, Pgpool-II[9], that distributes read queries to slaves and the master is used only for write queries. Moreover, Pgpool manages PostgreSQL configuration in the database servers and in case of master failover it automatically promotes a slave to master. In this way scalability and high availability is easily achieved.

To search among the stored metadata without needing to read the whole database contents, we use Apache SolrCloud [11], an indexer that allows full text search in the metadata. We created the cluster based on Apache ZooKeeper, which is responsible for the servers' coordination and synchronization. Each server contains all existing indices: one server is automatically elected as leader and it is responsible for writing new data while the rest servers replicate its data. Our application uses the Haystack API[10] to express modular search queries that can then be applied to whatever search engine Haystack is connected to. Because Haystack connects to a single Solr endpoint, an HAproxy load balancer for the Solr cluster was used here too ensuring scalability and high-availability for the Solr farm. A distributed SolrCloud cluster setup was also studied as an alternative in case of a great amount of metadata.

Detailed system deployment description

The infrastructure is deployed with the use of Ansible scripts. Scripts for both the initial setup and the later management of the infrastructure were created. The infrastructure

[8]http://www.haproxy.org/#desc
[9]http://www.pgpool.net/mediawiki/index.php/Main_Page
[10]http://haystacksearch.org/

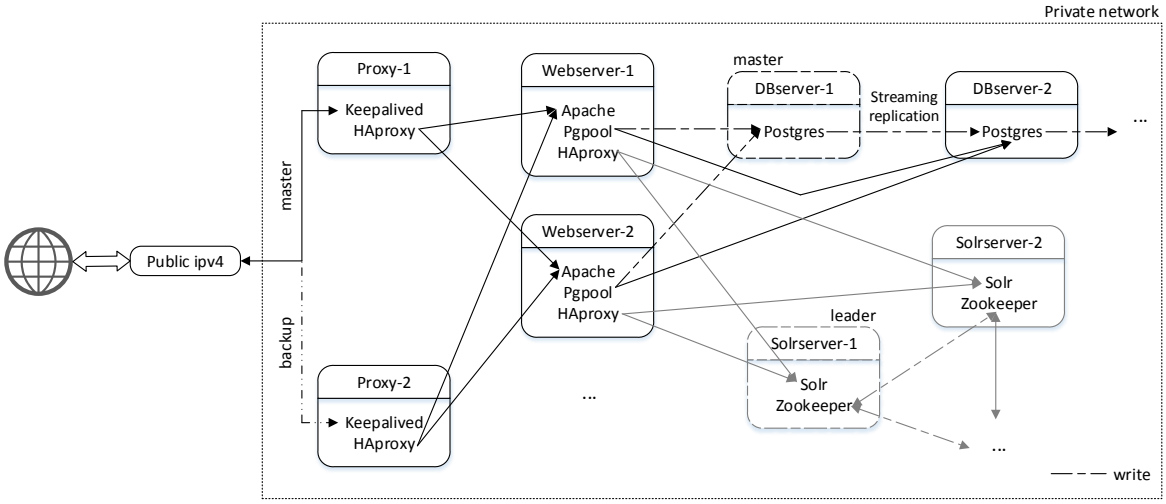

Fig. 2: Metadata storage and serving system's architecture

setup and management were tested in a cloud service named cyclades provided by the Okeanos IaaS [9]. For this purpose a library was built to be used by Ansible for accessing cyclades that uses kamaki, an Openstack compliant API.

The system architecture is shown in Figure 2. Virtual Machines (i.e., VMs) were created with the Debian operating system in a private network. We used a private network for internal servers' communication for safety reasons and because public IPs are generally scarce. Only one VM was given a public IP (matched to the site's DNS record), where the frontend HAproxy was installed. For high availability in case this VM fails, a backup VM with HAproxy was also created and Keepalived[11] was installed in both proxies (master and backup). In this way if one VM becomes unresponsive the other one is aware and automatically receives the public IP (running a user-defined script). HAproxy uses a list with the webservers' private IPs and distributes in a round-robin fashion the requests to the webservers using session stickiness [18]. The session stickiness attribute ensures that each user will be served by a single internal webserver throughout its entire session until he disconnects from the application (session management and storage on a database is required in the web application). Meanwhile, HAproxy automatically checks the webservers operation and uses a webserver only if it is responsive. This functionality ensures both high availability (in terms of an HAproxy or an internal webserver malfunctioning) and scalability (by easily and transparently adding more webserver VMs) for the entire web farm.

Regarding the webfarm, each webserver is a VM where an Apache http server is installed along with the web application. Pgpool-II, Monit[12] and HAproxy tools are also installed. Pgpool is used as a load-balancer for the database farm, as mentioned before. Monit is a monitoring and automatic management tool for system services that we configured to stop Apache and disable the webserver in case Pgpool is down. HAproxy is used as a load balancer for the Solr farm. We chose not to have the load balancers for the database farm and the Solr farm in separate VMs, but instead each webserver uses its own balancer since the frontend proxy guarantees load balancing until the level of the database and Solr farms. In this way we also avoided the complexity of adding failover detection for the VMs that would host these load balancers too. HAproxy has automatic failover detection as well as Pgpool, with the latest one allowing the user to perform extra actions through a user defined script. An important difference is that while HAproxy automatically reconnects operational servers, Pgpool does not support this automation yet. However, we implemented for this purpose an alternative procedure with scripts that are automatically executed by Pgpool in case of a failover. Scalability for the database farm and the Solr farm is also easily achieved by letting Pgpool and proxy know about the existence of new servers.

As previously mentioned, the database farm consists of VMs with PostgreSQL installed in master-slave mode. The master is configured to store in the WAL (write-ahead-log) files the data changes (up to 512MB) and the slaves are synchronized with the master using streaming replication. Streaming replication allows a slave server to stay up-to-date with file-based log shipping. The slave connects to the master, which streams WAL records to the slave as they are generated, without waiting for the WAL file to be filled. We set up cascading replication, i.e. each server opens a stream with the previous one and a chain of streams is formed for copying data. Alternatively all slaves could stream directly from the master but we avoided this implementation in order to minimize network traffic on the master. For the database we selected strong data consistency and the master responds to write requests only after the next slave that streams from it acknowledges that the data were written on its disk. There are of course more options for a weaker consistency that can be used in case of increased write workload. If a slave is disconnected and reconnects after a short period of time, it can synchronize its data if all the changes that it missed exist in WAL files. If it was disconnected for a longer period and the slave can not synchronize anymore its data must be rebuilt, i.e., transferred by the master.

Finally a VM was set up with Ganglia and Nagios moni-

[11] http://www.keepalived.org/

[12] https://mmonit.com/monit/#about

toring frameworks to monitor resources' usage metrics and the status of all services.

Scaling/updating the system

Each part of the infrastructure can be scaled out in case of increased user load. The addition of a new web/database/Solr server to scale out the infrastructure is manually performed with the use of a specific Ansible script that is implemented for each of the three server types.
Updating the web application version is also possible with the manual execution of the same script that setup the infrastructure initially. More Ansible scripts will be deployed in the future for updating the versions of the selected installed tools.

Recovery protocol

In this section we refer to all possible failure situations, how they are detected and the automatic or manual recovery mechanisms that are used in each case. We have three types of failures: service failure, network failure and VM failure.

1) Frontend proxies (both public and private IPs)

Failure detection: If in the master proxy the HAproxy service stops or the VM is destroyed or the private IP has connectivity issues (therefore there is no communication with the webservers) Keepalived automatically detects it. When this happens the backup proxy receives the public IP of the problematic VM within a few seconds ($<$10). When the private IP and HAproxy service function again normally in the master proxy it automatically receives again the public IP by executing the same script. Any connectivity issues with the public IP are not manageable since the site simply can not be accessed. In this case a possible solution would be the use of 2 public IPs for the site with round-robin DNS.

Recovery: If the HAproxy service stops or the VM is destroyed, manual actions are required in both cases. The service must be manually started or the VM must be rebuilt with the Ansible script that setup the infrastructure. Issues with the private IP are automatically fixed.

2) Webservers (private IPs)

Failure detection: If the Apache service stops or the VM is destroyed or the private IP has connectivity issues it is automatically detected by HAproxy and the problematic webserver is not used until it recovers.
If the Pgpool service stops, the webserver is not functional anymore since it can not communicate with the database farm. In this case Monit is configured to detect it and stops the Apache service too.

Recovery: If the Apache or Pgpool service stops or the VM is destroyed, manual actions are required in all cases. The services must be manually started or the VM must be rebuilt with an Ansible script that is used for webfarm scaling. Issues with the private IP are automatically fixed.

3) Database servers (private IPs)

Failure detection: If the PostgreSQL service stops or the VM is destroyed or the private IP has connectivity issues Pgpool automatically detects it in each webserver and the malfunctioning database server is no longer used. When the Pgpool detects a database server failure it automatically tries to perform some necessary actions with a custom shell script. With this script we make sure to establish a correct chain for cascading replication from the master VM to functional slave VMs or to promote a slave to master if the master VM failed. Pgpool does not detect automatically when a database server is functional again to start using it, but we have implemented an alternative procedure. In case of network issues due to increased traffic for example, we do not want slaves to be lost instantly or permanently. For this purpose, we built a script that is triggered after a failure happens, and it restores a slave to the Pgpool list of functional VMs if the connectivity issue is fixed within 5 hours and the malfunctioning slave can still synchronize with the master. If a slave is restored the script also fixes of course the chain for cascading replication.

Recovery: Issues with the private IP in a slave server are automatically fixed if the server restores within 5 hours. If more than 5 hours pass or there is an issue with the private IP in the master or the VM is destroyed or the PostgreSQL service stops, manual actions are required. In the first 3 cases the VM must be rebuilt as a slave with an Ansible script that is used for scaling the database farm and in the last case the PostgreSQL service must be manually started.

4) Solr servers (private IPs)

Failure detection: If the Solr service stops or the VM is destroyed or the private IP has connectivity issues HAproxy automatically detects it in each webserver and the problematic Solr server is not used until it recovers. If the leader VM fails then a slave is automatically promoted to leader through ZooKeeper.

Recovery: Issues with the private IP are automatically fixed. In case the leader recovers it is therefore used as a slave as another server becomes the leader. If the Solr service stops or the VM is destroyed, manual actions are required. The service must be manually started or the VM must be rebuilt with an Ansible script that is used for scaling the Solr farm.

B. Data Storage and Serving System

This subsystem is responsible for storing and serving the resources' bulk raw data. Raw data are stored by the web application in a cloud storage service using the appropriate adaptor, in our tests in pithos+ cloud storage provided by the Okeanos IaaS. The system can however be connected to any cloud storage service using the appropriate adaptor.

Storing/retrieving resources' raw data to/from pithos is done through user forms in the web application. The application in our case uses the kamaki API to connect with pithos and perform streaming upload or download. In order not to cause increased traffic to the network of the metadata storage and serving system from the raw data transfers performed by webservers, a second team of webservers was set up, the D/U (i.e., Download/Upload) servers, to be used exclusively for transferring data to/from pithos. D/U servers run the same web application with common webservers and they are normally connected to the database farm and the Solr farm. Their

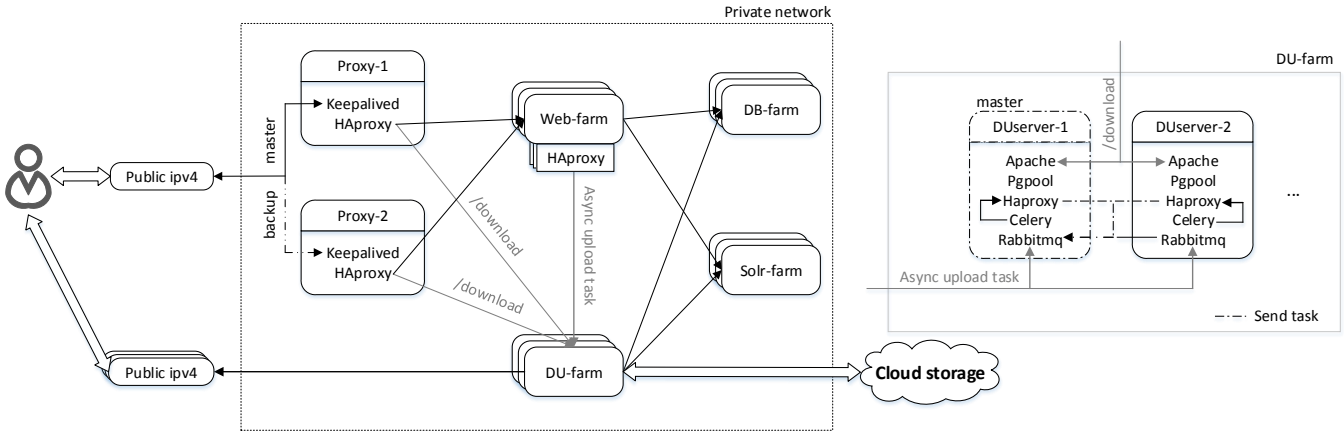

Fig. 3: Data storage and serving: DU-farm architecture and integration

difference is that they have their own public IP and DNS record offering a greater and scalable (proportional to the number of D/U servers) data transfer rate to/from pithos.

Because the data size may be in the order of GB or TB, data uploading is performed by the application asynchronously i.e. in a non-blocking way. For this purpose a task queue is used for uploads based on a message broker called rabbitmq[13]. All upload tasks are sent as messages to the rabbitmq queue and each message can be serially received by a celery worker[14]. A celery worker is a running thread ready to execute a function (or task) written in any language when it receives the relevant message. In our case we have a function that contains the python code that actually uploads the data.

This system's architecture is shown in Figure 3. In each D/U server we initialize multiple celery workers to enable multitasking and they all execute the upload tasks of a unique queue. To build a unique but highly available queue, a rabbitmq cluster was set up on top of the D/U servers in master-slave mode. The master is automatically elected among D/U servers and is in charge of distributing the queue's upload tasks to all the celery workers that serve the queue. The slaves forward incoming messages to the master and the master's queue is mirrored across all slaves for high availability. The webservers were then configured to distribute with their local HAproxy the upload tasks to the D/U servers' queues that are actually the same mirrored queue. In this way, if a D/U server fails the system will not be affected at all: a new master will simply be elected if needed and if any tasks have failed they will be re-executed in a user transparent manner. Finally, celery workers were configured to listen to messages not only from the master but from all D/U servers' queues, for high availability purposes (if the master changes they won't be affected).

The frontend load balancer (HAproxy) that distributes the requests to the webfarm, was configured to send download requests in a round-robin fashion to the D/U servers while preserving session information. D/U servers were configured to only serve the /download url for simplicity. In each D/U server Apache redirects the request to the server's own domain name so that the data is transferred through its public network interface and is not routed through the frontend proxy. Only registered users can therefore access the D/U servers for downloading since the local web application will check for the session existence in the common database. In this way we have scalability and high availability in the upload/download mechanism that we implemented too.

Scaling/updating the system

In case the system's network is stressed by multiple transfers performed by users the system can be scaled out. A new D/U server can be manually added with the use of a specific Ansible script that is implemented for scaling the DU farm. Updating the web application version is also possible with the manual execution of the same script that setup the infrastructure initially. More Ansible scripts will be deployed in the future for updating the versions of the selected installed tools.

Recovery protocol

D/U servers (public and private IPs)

Failure detection: If the VM is destroyed or the private IP has connectivity issues both the frontend HAproxy and the webservers' local HAproxy automatically detect it and the problematic D/U server is not used until it recovers.
If the rabbitmq service stops, the webservers' local HAproxy automatically detects it and upload tasks are not sent to the problematic D/U server's queue until it recovers. The rabbitmq cluster also automatically detects that the queue is not mirrored in this server but uploads and downloads can still be executed by the server.
If the Apache service stops, the frontend HAproxy automatically detects it and the problematic D/U server is not used for download requests until it recovers. In this case Monit is configured to detect it and stops the celery service too so the server does not execute upload tasks either. The server is still used for queue mirroring though.
If the Pgpool service stops, the D/U server is not functional anymore since it can not communicate with the database farm. In this case Monit is configured to detect it and stops the Apache service. The server is used only for queue mirroring in this case too.
If the celery service stops, the problematic server simply can't execute upload tasks anymore but other servers' workers will serve the queue's tasks.

[13]https://www.rabbitmq.com/features.html
[14]http://www.celeryproject.org/

Recovery: If Apache or rabbitmq or Pgpool or celery service stops or the VM is destroyed, manual actions are required in all cases. The stopped services must be manually started or the VM must be rebuilt with an Ansible script that is used for scaling the DU farm. Issues with the private IP are automatically fixed.

C. Distributed Processing System

This subsystem will be able to process data stored in the cloud. The infrastructure can be created on-the-fly or be in a stand-by mode: in the first case it will be destroyed after completing a task and free cloud resources. We selected the second mode and created a stand-by distributed Apache Hadoop cluster [22] with a predefined initial size and the available processing tools installed. The user will define an input resource in the form of a GUID and a processing tool from a list of available ones. The processing task will then be sent by the application in a rabbitmq queue available in the distributed processing system. Multiple celery workers are installed to handle this queue's tasks and each received processing task is translated by a celery worker to a Hadoop map/reduce job. In case of increased cloud resources usage, the infrastructure can easily be scaled out by adding extra nodes to the Hadoop cluster with the execution of an Ansible script.

The effective operation of the system is dependent on the data transfer rate to/from the cloud storage service. Since we are using large amounts of data, a Hadoop adaptor for the cloud storage system in use was integrated so that Hadoop can directly communicate with the cloud filesystem. In this way it can fetch and process chunks of a specific dataset in a parallel manner and achieve minimal execution time.

Also, taking into account that many users may require to process with the same tool the same dataset, the processing tasks' history is saved in a PostgreSQL master-slave database installed on this system. Before launching a new task each celery worker checks the database to avoid performing the same task again.

D. Automated infrastructure elasticity

Both the Metadata and Data Storage and Serving Systems as well as the Distributed Processing System are designed with an elastic architecture in mind. All the server farms included in the infrastructure can easily be scaled out with an execution of a specific procedure that was scripted with Ansible. The deployed scripts offer horizontal scalability by adding more servers to the infrastructure. Vertical scalability, i.e. increasing the cloud resources consolidated by each VM, could also be offered with automated scripts. However, we focused on horizontal scaling because there are limits regarding a VM's size in every cloud service and vertical scaling will not be possible at some point eventually. Also there is no difference between choosing vertical and horizontal scaling in terms of cost as long as total resources usage is the same.

The decision about whether a specific farm needs to be scaled out and at what extent, still must be taken by an application administrator. In case of increased load applied by users to any of the subsystems, the supervisor can monitor resources usage through the Ganglia platform and decide to scale out specific farms to cope with the load. This is of course not feasible in case of large scale content management systems that have a frequently varying load of requests (proportional to the number of concurrent users) as constant monitoring will be required in this case. Automated elasticity is therefore an important aspect that can be further investigated in a large scale deployment.

IV. Initial Performance Results

To demonstrate how the number of concurrent user requests affect the system's end-to-end performance and each individual component we executed a number of benchmarking tests on the metadata storage and serving system using the siege tool. Each database server was configured with 8 virtual CPUs (vCPUs), 8GB of RAM and 80GB disk space. Each web, Solr and proxy server was equipped with 4 vCPUs, 4GB of RAM and 10GB disk space. The infrastructure was initially deployed with two proxies, one web server, one database server and one Solr server to measure each single server's performance.

We executed simple HTTP GET requests on three different Urls for our experiments: the site's home page, a page that lists all available resources that are indexed by Solr and a page that displays the stored metadata information about a specific resource. We applied different loads by changing the number of concurrent requests made for each Url. 50 concurrent requests for example will be translated into 50 requests sent from different HTTP connections (i.e., threads) to represent 50 different users: whenever a request is answered a new one is dispatched by the same thread until the entire workload is executed. Each workload was applied for ten minutes on each Url. It is important to notice that the Apache disk cache was enabled in the webservers and that PostgreSQL and Solr by default cache answered queries in-memory, so in our experiments the first response for each url was cached and all following requests were served from the cache.

In our first experiment we applied a load of 50 concurrent requests on the home page. The system throughput was 1146.46 transactions/sec with an average response time of 0.04 secs. We noticed that the CPU usage on the webserver was constantly 23% during the workload execution. We then increased the number of concurrent requests up to 100 and then up to 200. In the 100 requests workload the throughput was slightly increased to 1213.75 transactions/sec. The CPU usage also slightly increased (26%) but the main difference on performance was that the average response time was doubled to 0.08 secs. In the 200 requests workload the performance was also increased to 1387.37 transactions/sec and the CPU usage to 30% but the average response time was nearly doubled again to 0.14 secs. The increased load applied by users has therefore an immediate impact mainly on the system's response time when the cache is used.

For the second page that retrieves all indexed resources from Solr (for a total of 600 resources) we applied two different workloads of 50 and 100 concurrent requests. For both workloads the webserver's CPU usage was constantly 20% and in the Solr server we noticed an initial CPU usage peak at 30% and then it remained very low (2%) since the result was cached. In the 50 concurrent requests case the throughput was 560.35 transactions/sec and the average response time was 0.09 secs. In the 100 concurrent requests case the throughput was decreased a little to 464.96 transactions/sec and the response

time was doubled to 0.19 secs. We notice in this case that the throughput was generally lower than in the homepage case. This is expected since the Solr farm is also involved in the request pipeline. The impact of increased input load was similar however as the response time was doubled.

For the third page that retrieves a resource's metadata from the database we also applied two different loads of 50 and 100 concurrent requests. The CPU usage was similar to that of the Solr index page except that the initial peak in the database server's CPU usage was observed at 50%. In the 50 concurrent requests case the throughput was 762.13 transactions/sec and the average response time was 0.06 secs. In the 100 concurrent requests case the throughput was increased a little to 898.78 transactions/sec and the response time was doubled to 0.11 secs. We also notice here the similar negative impact of the input load to the response time. The throughput was however higher than that of the Solr farm but still lower than that of the simple homepage as expected.

In the last experiment we added one more server to the web and Solr farms and two more servers to the database farm and applied the same workload. The results showed that the CPU load was equally distributed to the servers of each farm and each server exhibited half the CPU load compared to the previous execution. The throughput and response time remained the same as the system was not overloaded in our previous experiments due to the caching mechanism and the proxy simply offered the maximum possible transaction throughput in all our experiments. Both load balancing and scalability were therefore confirmed when we increased the system's resources. The throughput and response time improvement remains to be seen by applying a workload that includes a large number of different requests that can stress the system regardless of its caching capacity, or by completely removing the caching mechanism during the experiment execution.

V. Conclusion and Future Work

We have designed and presented in detail an architecture for a cloud-based content management system based on open source tools. Our system offers data serving, storing, indexing as well as processing functionality. Scalability and fault tolerance were thoroughly studied as our primary targets since we have selected cloud services as the best option for our implementation. The deployment and management of a system based on our architecture has been fully automated with Ansible scripts following the DevOps paradigm and was successfully tested in a cloud provider.

Regarding the architecture's flexibility to use various tools and be deployed in any cloud, the required changes are to update the ansible scripts and create a new library to be used as a cloud connector. The web application can also be configured to use another storage engine by using the appropriate connector. For example, the application can use Amazon S3 or Okeanos pithos+ for data storage, SolrCloud or Elasticsearch for data indexing and PostgreSQL or MySQL as the web application storage backend. The web application may be built using the Django or NodeJs frameworks and the deployment of the infrastructure can be done on various cloud services such as Amazon EC2, Okeanos cyclades or any Openstack compliant IaaS cloud. A thorough benchmarking for every application subsystem will be performed in the future for evaluation purposes in order to optimize their configuration.

Finally a mechanism for automated scalability will be studied to avoid constant monitoring and manual actions over the infrastructure. In [21] the authors present an open-source automated resource provisioning framework that could be integrated in our infrastructure. This framework could be used to scale out each server farm automatically based on resources usage by running the relevant Ansible script for scaling.

Acknowledgment

This work was partially supported by the European Commission in terms of the CELAR 317790 FP7 project (FP7-ICT-2011-8).

References

[1] Comparison of Open-source Configuration Management Software. http://en.wikipedia.org/wiki/Comparison_of_open-source_configuration_management_software.

[2] Comparison of Web Application Frameworks. http://en.wikipedia.org/wiki/Comparison_of_web_application_frameworks.

[3] Content Management Problems Need Big-Data Solutions. http://www.theserverside.com/feature/Content-management-problems-need-big-data-solutions.

[4] The Data Deluge. *The Economist*, Feb. 2010.

[5] G. Chowdhury and S. Chowdhury. *Introduction to Digital Libraries*. Facet publishing, 2002.

[6] A. Holovaty and J. Kaplan-Moss. *The Definitive Guide to Django: Web Development Done Right*. Apress, 2009.

[7] M. Httermann. *DevOps for developers*. Apress, 2012.

[8] I. Konstantinou, E. Floros, and N. Koziris. Public vs Private Cloud Usage Costs: the StratusLab Case. In *Proceedings of the 2nd International Workshop on Cloud Computing Platforms*, page 3. ACM, 2012.

[9] V. Koukis, C. Venetsanopoulos, and N. Koziris. okeanos: Building a Cloud, Cluster by Cluster. *IEEE internet computing*, 17(3):67–71, 2013.

[10] S. Krishnan. *Programming Windows Azure*. " O'Reilly Media, Inc.", 2010.

[11] R. Kuć. *Apache Solr 4 Cookbook*. Packt Publishing Ltd, 2013.

[12] S. Laumer, D. Beimborn, C. Maier, and C. Weinert. Enterprise Content Management. *Business & Information Systems Engineering*, 5(6):449–452, 2013.

[13] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, A. H. Byers, and M. G. Institute. Big data: The Next Frontier for Innovation, Competition, and Productivity. 2011.

[14] N. Middleton, R. Schneeman, et al. *Heroku: Up and Running*. " O'Reilly Media, Inc.", 2013.

[15] A. Qureshi, R. Weber, H. Balakrishnan, J. Guttag, and B. Maggs. Cutting the Electric Bill for Internet-scale Systems. *ACM SIGCOMM Computer Communication Review*, 39(4):123–134, 2009.

[16] A. Rockley, P. Kostur, and S. Manning. *Managing Enterprise Content: A Unified Content Strategy*. New Riders, 2003.

[17] D. Sanderson. *Programming Google App Engine*. O'Reilly Media, Sebastopol, CA, second edition, Oct. 2012.

[18] M. Stecca, L. Bazzucco, and M. Maresca. Sticky Session Support in Auto Scaling IaaS Systems. In *SERVICES*, pages 232–239, 2011.

[19] C. Thompson. *Smarter than you Think: How Technology is Changing our Minds for the Better*. Penguin, 2013.

[20] T. Tomlinson and J. VanDyke. *Pro Drupal 7 Development*. Apress, 2010.

[21] D. Tsoumakos, I. Konstantinou, C. Boumpouka, S. Sioutas, and N. Koziris. Automated, Elastic Resource Provisioning for NoSQL Clusters Using TIRAMOLA. In *CCGrid*, 2013.

[22] T. White. *Hadoop: the Definitive Guide*. " O'Reilly Media, Inc.", 2009.