Santiago Durán, M., González Compean, J.L., Brinkmann, A., Reyes Anastacio, H.C., Carretero Pérez, J., Montella, R., Toscano Pulido, G. (2020). A gearbox model for processing large volumes of data by using pipeline systems encapsulated into virtual containers. *Future Generation Computer Systems*, 106, pp. 304-319.

DOI: 10.1016/j.future.2020.01.014

A gearbox model for processing large volumes of data by using pipeline systemsystems encapsulated into virtual containers

Miguel Santiago-Duran [a], J.L. Gonzalez-Compean [a,*], André Brinkmann [d], Hugo G. Reyes-Anastacio [a], Jesus Carretero [b], Raffaele Montella [c], Gregorio Toscano Pulido [a]

[a] Cinvestav Tamaulipas, Mexico [b] University Carlos III of Madrid, Spain [c] Parthenope University of Naples, Italy [d] Johannes Gutenberg University Mainz, Germany

a b s t r a c t

Software pipelines enable organizations to chain applications for adding value to contents (e.g., confidentially, reliability, and integrity) before either sharing them with partners or sending them to the cloud. However, the pipeline components add overhead when processing large volumes of data, which can become critical in real-world scenarios. This paper presents a gearbox model for processing large volumes of data by using pipeline systems encapsulated into virtual containers. In this model,
the gears represent applications, whereas gearboxes represent software pipelines. This model was implemented as a collaborative system that automatically performs Gear up (by using parallel patterns) and/or Gear down (by using in-memory storage) until all gears produce uniform data processing velocities. This model reduces delays and bottlenecks produced by the heterogeneous performance of applications included in software pipelines. The new container tool *capsule* has been designed to encapsulate both the collaborative system and the software pipelines into a virtual container and deploy it on IT infrastructures. We conducted case studies to evaluate the performance of *capsule* when processing medical images and PDF repositories. The incorporation of a capsule to a cloud storage service for pre-processing medical imagery was also studied. The experimental evaluation revealed the feasibility of applying the gearbox model to the deployment of software pipelines in realworld scenarios as it can significantly improve the end-user service experience when pre-processing large-scale data in comparison with state-of-the-art solutions such as Sacbe and Parsl.

Ubiquitous and pervasive technologies based on cloud services have become common solutions for organizations to manage, preserve and share large volumes of data in a cost-effective manner [1,2]. Nevertheless, organizations might lose control over their data when either sharing information with partners or outsourcing the data management to cloud providers. This lack of control has resulted in incidents affecting business continuity by outages [3], violating confidentiality and integrity of data [4] and leading to unauthorized accesses [3] or data accumulation [5].

In order to profit from economic benefits when using ubiquitous and pervasive technologies without losing control over data, prevention actions are required to mitigate the risks that are still associated with this type of technology [6,7]. End-to-end pre-processing solutions have therefore been proposed to cope with the lack of control over data. For instance, encryption solutions are used to reduce security issues [6,8–10], coding and redundancy data schemes to overcome the effects of outages or failures [11–13], compression tools to reduce the costs of data accumulation in the cloud [14], and acceleration strategies based on parallelism to minimize latency effects during data transfer [15,16]. In general, it holds that content pre-processing by a given application can add a control feature/property to that content. This enables an organization to mitigate a given risk that eventually could arise in the cloud.

In real-world scenarios, organizations often require adding several orthogonal control features to their data, depending on either the sensibility or importance of and the operations to be performed with the data. For instance, reliability and costefficiency are desirable features to be added to data that will only be uploaded/downloaded in/from the cloud, whereas security plus reliability and cost-efficiency features should be added to data when managing sensitive data. Moreover, data sharing scenarios require additional control features like integrity to avoid/discover content alterations and authenticity based on access control to ensure that only valid users are getting access to the contents and to avoid non-repudiation by end-users participating in these operations.

1. Introduction

To simplify the building and management of these heterogeneous solution scenarios, pipelines of applications built by using pipe&filter patterns [16,17] have been proposed to add different control features to the contents [18–21].

However, the implementation of pipelines in real-world scenarios can produce idle times and bottlenecks based on the data exchange between the applications included in a pipeline system. Their main cause is the heterogeneous performance of the applications being composed within the pipeline. In short, slow applications produce idle times in the rest of the applications by not being able to process enough data for the following pipeline stages, while fast applications lead to bottlenecks in slow applications by pushing data out to fast.

The load imbalance is expected to increase when scaling the number of applications in a pipeline and when processing large volumes of data, which is becoming a common scenario in enterprise environments. For instance, organizations and even governments are imposing restrictive rules for protecting contents such as satellite data [22] and medical images [23], which are considered very sensitive data that must be preserved for long periods of time. When attending and observing these rules, an accumulation of contents arises and organizations end up processing large volumes of data [19,24]. In practice, this accumulation becomes critical for organizations because the processing tasks must be performed efficiently to avoid impacting the service experiences of end-users.

In this context, organizations not only need solutions to create pipeline systems for adding control features to the contents but also it is required that these solutions offer mechanisms for processing large volumes of data in an efficient manner.

This paper presents a container-based processing model to build application pipelines for organizations to efficiently process large volumes of data. This model is based on a gearbox metaphor where gears represent applications, whereas boxes represent pipeline systems. In a gearbox model, a workload manager based on a collaborative system automatically performs gearing up (by using parallel patterns) and/or gearing down (by using in-memory storage) procedures until all applications in a pipeline produce as uniform data processing velocities as possible. The manager has been developed as a software component called *capsule*, which represents a virtual container image that executes the pipeline systems built by organizations but modeled as a Gearbox.

These capsules were evaluated through experiments processing data from both medical imagery and PDF repositories and were compared with traditional engine services and state-of-theart solutions.

A study case was also conducted based on an implementation of a pre-processing pipeline system in the form of a capsule, which was incorporated to a cloud storage service for supporting medical imagery repository management.

The main contributions of this paper are:

- A novel gearbox model for software pipeline systems encapsulated into virtual containers: It enables organizations to create application pipelines adding properties to contents in the form of gearboxes.
- A collaborative system to implement gearboxes in real-world scenarios: It enables organizations to deploy software pipelines in virtual containers for processing large volumes of data in an efficient manner.

The rest of the paper is organized as follows. The gearbox model is presented in Section 2. The design principles of a gearbox are described in Section 3. The evaluation methodology and experimental results from experiments and case studies are described in Section 4. The performance results are described in Section 5. The study case based on the incorporation of a capsule into a cloud storage service is presented in Section 6. In Section 7 the related work is described. Finally, conclusions and future research lines are described in Section 8.

2. A gearbox model for the management of software pipelines encapsulated into virtual containers

In this section, we define the key concepts of the Gearbox metaphor, the gearbox model. We also describe the development and implementation of pipeline systems encapsulated into virtual containers based on this model.

2.1. Gearbox metaphor concepts

A traditional gearbox system consists of a series of gears integrated into a box to produce a given velocity.

The small gears increase the velocity in a gearbox, whereas big gears reduce velocity. The difference of velocities created by the size of gears is expressed as a ratio. Knowing the ratios, the designers can create a gearbox layout to achieve a given velocity through the box by performing gearing up/down procedures.

In this metaphor, ' ' gears represent a ' virtual computing space for filters (applications), whereas gearboxes represent a virtual computing space to be used by the pipeline systems. The basic idea is to homogenize the ratios of all pair of gears (filters/applications) by performing gearing up and down procedures, which will be described in next section.

The following are the logical components considered in Gearbox metaphor:

- *Pipes*&*Filters*: A Filter represents an application, whereas a Pipe represents either an input or output interface.
  Filter ⇔ *Application*
  Pipe ⇔ *I/O* interface
- *Pipeline* is a processing structure where filters are interconnected in a sequential manner by using pipes. In this structure, an incoming data is delivered to a filter (by using an input pipe), which transforms it into a new version that is delivered to next filter by using an output pipe. This process is repeated until the last filter in a pipeline delivering data to a data sink. (see an example in Fig. 1).
- *Gearbox* is the logical representation of a software pipeline built by using the gearbox model. A *Gearbox* includes *GearArray* and *GearboxCS*.
- *GearArray* is a series of gears included into a Gearbox:
  $\{Gear_i, Gear_{i+1}, \ldots, Gear_{i+n}\} \in GearArray \in Gearbox$.
- *Gear* is a logical structure including one filter (application), the Input and Output pipes (I/O interfaces) as well as a *middleware* for the communication of this structure with a collaborative system called *GearboxCS*. A gear thus follows an Extract (from $Pipe_{in}$), Transform (by using the filter) and Load data (to the $Pipe_{out}$) model.
- *GearboxCS* is a collaborative system in charge of the management of the communication and data exchange within a software pipeline by using the principles of Gearbox model.

The *middleware* instances and the components of *GearboxCS* will be described in detail in Section 3.1.

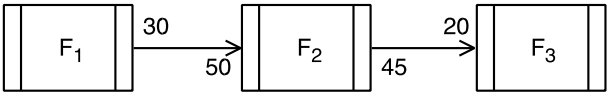


Fig. 1. Conceptual example of retrieval throughput (RT) and delivery throughput (DT).

2.2. Gearbox model formalization

In previous works, we designed pipeline-based solutions for a space agency [19], medical [24] and regular organizations [6] to add value to their contents before sharing data with partners or sending these contents to the cloud. In this process, we found that applications with small service times (fast filters) created bottlenecks when being coupled with slow applications, whereas slow ones led to idle times for fast ones. It is therefore desirable that the filters' service times are either as homogeneous as possible (which is not to be expected) or that at least the throughput of each filter stage is comparable.

Fig. 1 shows an example of a software pipeline including applications with different input and output performance. We call *retrieval throughput* (*RT*) to the input performance, which is measured by the amount of data read per unit of time, whereas we call *delivery throughput* (*DT*) to the output performance that is measured by the data processed per unit of time. It must be noted that different retrieval and delivery throughput can only be achieved by using buffers within a filter component.

In this example, applications produce different retrieval/ delivery throughput; as a result, a performance mismatch is evident. As it can be seen, the fastest application (represented by *F*2) suffers idle times by waiting for *F*1 when it is delivering results and produces queuing in *F*3, which also produces delays in results sent from *F*2 to *F*3. In real-world scenarios, this behavior represents a challenge for developers specially when processing a large set of data.

To solve this type of problem, we designed a model based on a gearbox metaphor. We recall that, in this metaphor, gears represent a virtual computing space for filters (applications), whereas gearboxes represent a

virtual computing space to be used by a software pipeline (see the concepts of Gear and Gearbox previously defined in this paper).

In a traditional gearbox system, the small gears increase the velocity in a gearbox, whereas big gears reduce velocity. The difference of velocities created by the size of gears is expressed as a ratio. Knowing the ratios, the designers can create a gearbox layout to achieve a given velocity through the box by performing gearing up/down procedures.

The basic idea is monitoring the service times of the applications in a software pipeline to determine the retrieval/delivery throughput of gears. The gearbox model uses this information to calculate the size of the gears and the ratio of each pair of gears in a gearbox. This information represents the input parameters of gear up/down processes.

As we already said, the gear up process is performed in gears producing slow input/output performance. This process is implemented by using parallel patterns. In this model, a gear up is achieved by cloning gears and deploying them on a virtual container by using control structures, which organize the gear clones in the form of parallel patterns that are managed in this model as a single virtual gear. These gear clones process data in parallel producing an acceleration effect that improves the retrieval/delivery throughput of a slow gear.

Fig. 2 shows a conceptual representation of a software pipeline system modeled as a gearbox for the scenario shown in Fig. 1 (for *F*1 and *F*2). In this example, the slow application represented by *F*1 is cloned to be used in parallel within the gear abstraction (gear up).

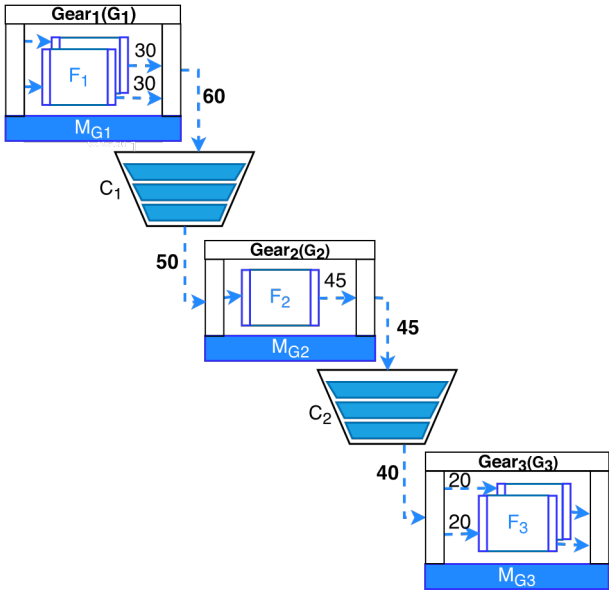


Fig. 2. Conceptual representation of a pipeline system modeled as a gearbox.

In turn, a gear down process is achieved by using in-memory storage based on virtual buckets placed within a computing space of the gearbox. This process artificially reduces the delivery throughput of a gear by temporally storing, in-memory, the surplus delivered data. This avoids a gear to delivery workload that the next gear cannot process.

Fig. 2 shows that the fast application represented by *F*2 is connected to a bucket system by using in-memory storage and managed within the gear abstraction. This enables the pipeline system to absorb the surplus data produced by F2 (gear down) and to match it with the input performance of the next application *F*3.

The methodology for modeling a software pipeline system as a gearbox includes three phases:

1. In the measuring phase, the data retrieval/delivery throughput of gears are captured in execution time by measuring the amount of data passing through the input/output interfaces of each application in a pipeline system. This enables the model to calculate the gear sizes of each application.

2. In the temporizing phase, a gearbox layout is created when modifying the size of the gears by performing gear up/ down adjustments until the gear ratios are balanced. The gearbox model homogenizes the retrieval/delivery data throughput by adjusting the size of the gears, which balancing the ratios of all the gears in a gearbox. This procedure is performed until the differential among the gear velocities is minimized, which will yield a nearly constant dataflow through the software pipelines that will reduce bottlenecks and idle times in the applications of a software pipeline.
3. In the deployment phase, the gearbox configuration layout is implemented in the software pipeline by a subjacent collaborative system (*GearboxCS*). This system applies the gear up/down changes suggested in the gearbox layout to the pipeline system. At this point, the collaborative system can serve the workload to be attended by the pipeline system based on the gearbox layout configuration. Adjustments in the gearbox layout are performed over time, in execution time, depending on the changes applied to the pipeline system (e.g., by adding/removing filters to/from the pipeline).

It is expected that all the systems considered to build a gearbox are encapsulated into a virtual container for all systems to share resources.

2.3. The measuring phase of data retrieval and delivery throughput of the gears

In this section, we describe the measuring phase of the methodology that we defined for modeling a software pipeline as a gearbox.

We are assuming that the developers have fixed an amount of RAM and number of cores to be used in the virtual container. The gears considered in a *Gearbox* therefore get resources from the virtual container on-demand. This means the fast gears get access to more resources than slower of *k* gears ones. In this scenario, the velocity of $\{Gear_1, Gear_2, ..., Gear_k\} \in Gearbox$ is measured by the retrieval and delivery throughput of the gears (RT_k and DT_k). These metrics are captured each time a gear processes a sample of a fixed number of requests (*Sa*).

The mean *RT* metric for a sample of contents *Sa* is calculated by using the following equation:

$$MRT_{Geark} = \frac{\sum_{Sa} |C_i| \, ST_{Ci}}{Sa} \quad (1)$$

The *ST* represents the service time spent by $Gear_k$ to acquire and process each content (C_i) in an analyzed sample (*Sa*).[2] This metric is used to determine the gear representing a bottleneck (*GB*) for the pipeline, if any. This is determined by the following formula:

$$GB = \min\{RT_{Gear1}, RT_{Gear2}, ..., RT_{Geark}\} \quad (2)$$

The mean delivery throughput (*DT*) is calculated by the formula used to calculate *RT* but using the data passing through the output interface of a gear and it is used to determine the fastest gear (*FG*) in a Gearbox by using the following formula:

$$FG = \max\{DT_{Gear1}, DT_{Gear2}, ..., DT_{Geark}\} \quad (3)$$

In this model, the mean *RT* represents the demand of a gear, whereas mean *DT* represents its production. A data structure ($RDT_{DT,RT}$) is created when *DT* and *RT* metrics have been calculated for all the *gears* ∈ *Gearbox*. This structure is used as input parameter for the gear up process.

Notice that both metrics could report different values depending on the input or output interface used by a gear. For instance, *RT* and *DT* of gears either connected to data sources or data sinks may be different as the input or output interfaces of those gears to either data source (file system) or remote location through network (cloud location) respectability instead to the shared memory.

In such a scenario, this gear produces a throughput imbalance when the input is retrieving data from a data source, whereas the output delivers data to shared memory. It also happens to the delivery throughput when a

[2] Median or mode can be used to adjust this metric when the virtual container is sharing resources with other virtual containers.

gear retrieves data from an input interface through the shared memory, but the output is delivering data to a slower data sink. In both cases the I/O costs are different for the input and output interfaces.

2.4. Temporizing phase; creating gearbox layouts

In this section, we describe a temporizing method based on gear up and down procedures.

We identified two possible methods to homogenize the velocities (DT and RT) of all the Gears. The first one is to solve the bottleneck (*GB*) by performing Gearing up over this very gear and then identifying the new *GB* to perform a new gearing up in an iterative procedure that finish when using all resources of the virtual container of the capsule. The second one is to compare the *DT* and *RT* of each gear with the fastest gear (*FG*) and to calculate the parameters of the gear up process for each gear. This process is performed until the *RT* and *DT* of all gears are either as homogeneous as possible or at least comparable to the *FG*. We used the second method to perform the temporizing of the gears in a Gearbox.

The first step in a gear up process is to calculate the ratio of the relation demand/production of each pair of gears in a capsule (*RatioPG*), which is calculated depending on the *RT* and *DT* metrics of a pair of gears.

$$RatioPG = \begin{cases} \left\lfloor \frac{DT_k}{RT_{k+1}}, \right\rfloor & DT_k > RT_{k+1} \\ \left\lceil \frac{RT_{k+1}}{DT_k}, \right\rceil & RT_{k+1} > DT_k \end{cases} \quad (4)$$

RatioPG represents the performance difference between a slow (*Driven*) and a fast (*Driver*) gears in a pair of gears. *RatioPG* is used to determine the number of clones (*RClones*) recommend for a *Driven* gear to produce a comparable velocity to its *Driver* gear. The *RClones* metric must be calculated for each pair of gears in a Gearbox and starts with the pairs of gears including *FG* gear.

Nevertheless, the number of clones (*RClones*) should be limited by the resources assigned to that Gearbox. The number of cores (*NCores*) is the metric considered in this situation because RAM and storage are shared by all the gears (clones of driven included) considered in a Gearbox. This avoids the model inducing overload to the pipeline system. We are considering pessimistic and optimistic strategies to avoid this issue. The pessimistic one assumes that the limit to deploy clones on a Gearbox is the number of physical cores (*NCores*) assigned to the virtual container. In turn, the optimistic one includes both physical and virtualized cores in this calculation, which enables the model to increase the number of gear clones in a Gearbox.

$$CRU = \frac{CClones + RClones}{NCores} \quad (5)$$

The utilization of resources in a virtual container (*CRU*) is determined by the current clones (*CClones*) plus the recommended clones (*RClones*) in comparison with the available cores (*NCores*) in the virtual container used by a Gearbox. This enables the system to keep the resource demand of the clones in the limit of resources assigned to a gearbox.

When the gearing up procedure is finished, the gear down is executed whenever the model not finding a layout where all gears reaching a *DT* comparable to the *DT* of the *FG* gear. This process is applied to fast gears to reduce the mean *DT* of these gears in an artificial manner. The gear down is based on the virtual bucket system deployed on in-memory storage where an amount of memory ($AM_{k,k+1}$) is available for each pair of gears. This metric represents a limit to be used by the Gears and it can be either fixed at configuration time by using a given constant memory capacity or the gearbox can assign it on-demand. The utilization of memory per gear ($MU_{k,k+1}$) is used to calculate the availability of memory to create virtual buckets. This metric can be calculated by a simple difference between the sum of delivered data (*DD*) by $Gear_k$ and the sum of retrieved data (*DR*) for the requests to be processed (*nrtp*). At this point, it is expected $DT_k \geq RT_{k+1}$ as the driven and drive gears have been identified (see an example in
Fig. 3).

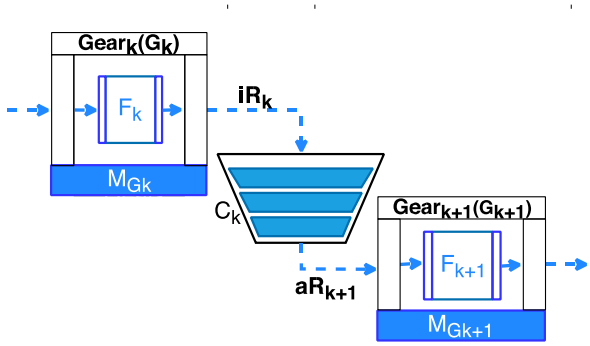


Fig. 3. In-memory storage bucket system for fast gears.

$$MU_{k,k+1} = 1 - \frac{\left[\sum_{r=1}^{nrtp} DT_k\right] - \left[\sum_{r=1}^{nrtp} RT_{k+1}\right]}{AM_{k,k+1}} \quad (6)$$

In scenarios where the model starting a gearing down procedure, $AM_{k,k+1}$ can be increased in execution time to reduce the value of $MU_{k,k+1}$ or to keep $MU_{k,k+1} < 1$.

A gearbox layout is created when the ratios of all gears have been homogenized by using both gear up/down procedure.

3. Gearbox model development: Design principles

In this section, we describe the design principles followed for implementing the gearbox model in the form of a collaborative system named *GearboxCS*, which represents the last phase of the gearbox implementation methodology.

GearboxCS builds pipelines, implements gearbox layout configuration to manage pipelines as a gearbox, and manages the delivery/retrieval workload to/from the software pipeline.

3.1. Gearbox design: concept definition

In this section, the key concepts of the Gearbox design are described in detail.

Fig. 4 will be used to depict the interconnections of the following virtual components considered in the Gearbox design principles:

- *Middleware*, included in each gear, was developed in the form of a pipeline engine that launches applications and to couple a gear with other one in a gearbox (pipeline). A middleware thus represents an intermediary between a pipeline filter and *GearboxCS*. See {M_{Gi}, M_{Gi+1}, M_{Gi+n}} in Fig. 4.
- *GearboxCS* is a collaborative system that manages the messages and data exchange within the software pipeline as well as applies the principles of Gearbox model to the software pipeline management. *GearboxCS* includes components such as *Workload Dispatcher, In-Memory Storage Service, SyncSystem and Manager*.
- *Workload Dispatcher* manages the input/output operations arriving to the pipeline of a Gearbox. This dispatcher creates, in an *In-Memory shared Storage Service*, areas for the exchange of data within a Gearbox. This service is used by two components: The middleware instances that invokes this service to intercept the I/O data arriving/departing to/from the software pipeline as well as the Syncs associated with each gear, which invoke this service for distributing workload to the clones of the gears ($Gear_i + 1$, $Gear_i + n$).

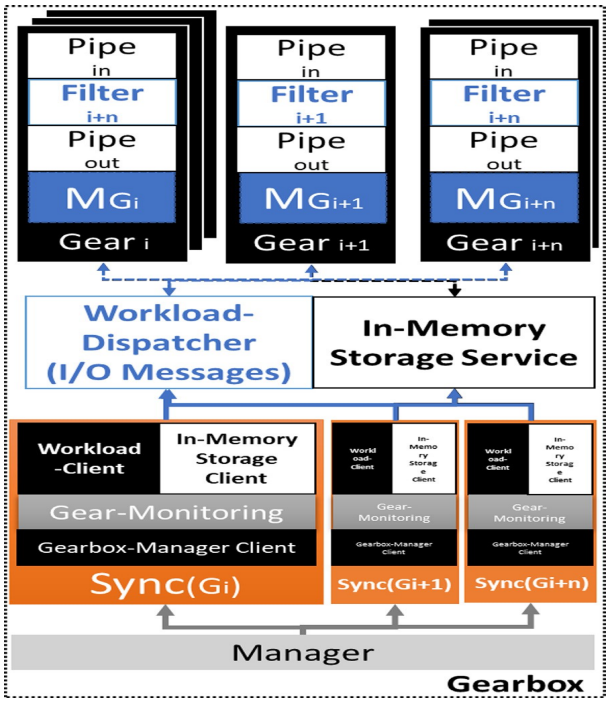


Fig. 4. A collaborative system for adapting pipeline systems to the gearbox model.

- *In-Memory Storage Service* is a shared memory service that enables *Workload Dispatcher* to create resource shared patterns for the gears to get data and put results required and produced by corresponding filters. These exchange areas also enable the components of *GearboxCS* to exchange messages and/or data each other.
- *SyncSystem* is a collaborative synchronization system including components named Syncs (see *Sync*(G_i), *Sync*($G_i + 1$), *Sync*(G_n)) in Fig. 4).
- *Sync* implements *Gearing Up* and *Gearing down* to face up bottlenecks and to yield a dataflow as continuous as possible. This means that a *Sync* creates clones of its original gear for implementing gearing up procedure. *Sync* also manages the I/O in gearing down procedures performed in its gear by using the *In-Memory Storage Service*.

A *Sync* includes components such as *Workload Client, Inmemory Storage Client, Gear-Monitoring, and Manager-Client*. See a zoom in of *Sync* (*Gi*) depicting its components in Fig. 4. • *Workload Client* is a client of *Workload Dispatcher* service. This client adapts the workload of a gear to the changes performed in the Gearbox layout by the *Manager*. This client thus distributes the load arriving to a gear to the clones of that gear.

The Workload Dispatcher client is used by the Syncs to distribute pointers of the shared memory to the gears. The gears use these pointers to get/put data/results from/to the shared memory. When a gear delivering results, it also puts a message of finalization in the shared memory for the dispatcher can deliver new pointers to that gear. This pointer delivery scheme based on messages through the shared memory produces a load balanced in the gearbox as it keeps all the gears processing load without calculating the utilization/consumption of the resources when making decisions on data distribution.

- *In-Memory storage Client* is used by *Syncs* for adapting the workload within a gear to gearing down procedures by getting/putting data and/or messages from/to shared memory.
- *Gear-Monitoring* keeps a register of delivery and retrieval rates of a gear. This software captures the *RT* and *DT* as well as creates an array of these metrics (called *I/O − trace*) produced by a fixed sample of requests (*r*). It also keeps the parameter that determines the number of requests to be included in *I/O − trace* (it is set at configuration time).
- *Manager-Client* enables the *Syncs* to deliver data to the *Manager* of the Gearbox and to receive information about the changes performed by the *Manager* in the Gearbox layout.

- *Manager* creates a gearbox layout (big picture) by executing the procedures of Gearing up and down, which are performed in execution time by using data produced by all the Syncs in the Gearbox. The communication of Manager with its Syncs is performed through a shared resource pattern, which means that the Syncs send/retrieve their information to/from the in-memory storage service, where the manager can get it. This method is used also in the communication between the gears and the Syncs.

3.2. Operation management in Gearbox

The operation of this collaborative system starts with the launching of one *middleware* instance for each gear defined to execute an application in a pipeline. The *middleware* intercepts applications' I/O requests (from $Pipe_{in}$) and delivers the results to the *In-Memory storage* service through the *Workload Dispatcher*. At this point, the collaborative system is considering that each *middleware* in the pipeline has launched one *Sync* software instance for retrieving, from In-Memory storage service, the *I/O − trace* produced by the *middleware*. This instance calculates the mean of *RT* and *DT* (*RDT*) by using the *I/O − trace*.

Algorithm 1 describes the procedure performed by the *Syncs* to calculate the *RDT* of the gears.

The instance of software called *Manager* collects the *RDT* produced by the *Syncs* to perform the gear up/down procedures.

Algorithm 2 has been proposed for the *Manager* to determine the number of clones required to balance the *RDT* of the gears in gearing up/down processes and then to create a gearbox layout.

Algorithm 3 shows a procedure for the Manager to maintain the gearbox layout in the limits of resources assigned to a Gearbox. It is also useful in scenarios where either the resources of the Gearbox are modified or launched it in a different infrastructure from the original one where that Gearbox was deployed on.

The Manager creates a new gearbox layout by using this information and informs the *Syncs* about changes to be performed in that configuration. Each *Sync* reads, from the In-Memory storage service, the messages sent by the manager and applies the changes suggested by the Manager to its gear.

Notice that the changes suggested by the *Manager* are performed by the *Syncs* without modifying the code of the applications by only using the *Middlewares* of the gears, which

Algorithm 1: Calculating mean throughput (RT_k and DT_k) for a slot of time.

```
Input  : nSyncs: Number of Syncs, AC: address control, time: time to wait for the completion of samples of RTk and DTk, nsamples: Number of samples to be
         considered in the throughput calculation.
Output : RDT: array of RTk and DTk means a set of samples.
1  RT[nSyncs]; DT[nSyncs]; RDT[2];isFinished[nSyncs];
2  allFinished = FALSE;
3  while !allFinished do
4  sumretrv ← 0
5  sumdel ← 0
6  setfalsevalues(isFinished)
7
8  sleep(time)   for k ← 0; k < nSyncs; k + + do
9                if AC[k].processedSamples = nsamples & !isFinished[k] then
10               samples ← AC[k].samples
11               for i ← 0; i < nsamples; i + + do
12 retrv ← samples[i].inputFlowSize /
                          samples[i].serviceTime
13 del ← samples[i].outputFlowSize /
                          samples[i].serviceTime
14 sumretrv ← sumretrv+ retrv
15 sumdel ← sumdel+ del
16               end
17               RT[k] ← sumretrv/nsamples
18               DT[k] ← sumdel/nsamples
19               isFinished[k] ← TRUE;
20           end
21 end
22 allFinished ← checkfinishedstatus(isFinished)
23 end
24 RDT[0] ← RT
```

```
25      RDT[1] ← DT
26      return RDT
```

Algorithm 2: Calculating the number of clones per gear (ClonesGear).

```
Input : nSyncs: Number of Syncs, RT: array of averages for all the nSyncs gears {RT_1, RT_2,..., RT_nSyncs}, DT: array of averages for all the nSyncs gears {DT_1, DT_2,..., DT_nSyncs}.
Output : ClonesGear 1 ClonesGear[nSyncs]
2  position ← maxValueIndex(RT)
3  if position > 0 then
4      for i = position; i > 0 ; − − do
5          ClonesGear[i − 1] ← ceil(RT[i] / DT[i − 1])
6          if ClonesGear[i − 1] = 0 then
7              ClonesGear[i − 1] ← 1
8          end
9      end
10 end
11 if position < nSyncs then
12     for i = position; i < nSyncs −1 ; + + do
13         ClonesGear[i + 1] ← floor(DT[i] / RT[i + 1])
14         if ClonesGear[i + 1] = 0 then
15             ClonesGear[i + 1] ← 1
16         end
17     end
18 end
19 ClonesGear[position] ← 1
20 return ClonesGear
```

intercept and manage the I/O requests of the applications by using *Workload Client*. This ensures that the workload management is automatic and transparent to both the applications and the pipeline system.

3.3. Managing clones in parallel patterns

When a *Sync* receives the order from the Manager to clone a filter, the *Sync* not only launches the clones into the gear **Algorithm 3:** Adapting the clones to the resources assigned to the capsule.

```
Input  : nSyncs: Number of Sycns, NCores: Number of physical cores assigned to the capsule, CClones: Number of clones currently used in a gearbox layout.
Output : nCF: Number of clones per filter. 1 RClones ← 0
2  for k ← 0; k < nSyncs; k + + do
3      RClones ← RClones + CClones[k]
4  end
5  while RClones > NCores do
6      aU ← (RClones − NCores) / RClones
7      for k ← 0; k < nSyncs; k + + do
8          nCF[k] ← CClones[k]  ceil(CClones[k] ∗ aU)
9          if nCF[k] < 1 then
10             nCF[k] ← 1;
11         end
12     end
13     RClones ← 0
14     for k ← 0; k < nSyncs; k + + do
15         RClones ← RClones + nCF[k]
16     end
17 end
18 return nCF
```

environment but also executes a software instance of *Workload Client* to keep control over the Inputs/Outputs produced by the clones of a Gear.

This software instance organizes the clones of gears launched in a gearbox in the form of a parallel pattern. It also establishes controls over the synchronization of the workload, data delivery/retrieval and distribution of load to be sent to the clones in a balanced manner.

This instance re- uses the *Middleware* of the gears as workers. This software instance therefore intercepts the load sent to the original gear and performs an indirection to the clones; as a result, the clones process load in concurrent manner. This scheme produces task parallelism in each gear where it is used.

3.4. Resource shared patterns for gears to exchange information

All the software instances considered in the gearbox model creates resource shared patterns in the in-memory storage service.

This means that the gears and control instances share memory pointers created by using the in-memory storage service when exchanging data with each other, which is not removed from memory. This avoids the usage of expensive I/O calls (e.g. file system and network) in the collaborative system.

In the example shown in Fig. 4, the first *Gear* reads content to be processed from a hard disk (data source), it invokes the in-memory storage service to place that content in the shared memory. In this way, the gear can get access to data from the memory instead of the hard disk or a cloud location. The results produced by $Gear_i$ are intercepted by its middleware and placed in-memory, the pointers of the memory are delivered to the middleware of the next gear ($Gear_i + 1$), which basically gets the content from the shared memory.

Gearbox also provides gears with file system interfaces to retrieve contents from a data source and to deliver them to a data sink, which is the case of $Gear_i$ and $Gear_i + n$.

3.5. Development of software pipelines based on the gearbox model

In this section, we describe the strategy used to encapsulate software pipeline systems built by using gearbox layout configurations into virtual containers.

3.5.1. Gearbox Capsule image

The collaborative system used to develop a Gearbox depicted in Fig. 4 is implemented by using a deployment structure called *Gearbox Capsule*, which basically is a virtual container image.

This Capsule image includes the following software instances: a reduced operating system, the Gearbox components previously described as well as all the dependencies and libraries required by these components. This image also includes I/O configuration file which set the network and file system interfaces to be used by Gearbox Capsule to extract data from the source and to load the results produced by the Gearbox in, for instance, a storage sink.

A *Capsule* therefore is created as a container configuration file (e.g. a Dockerfile), which grants that the pipelines can be deployed on an infrastructure (any of server, cluster or cloud) having installed a container platform (e.g. Docker). This development structure converts a software pipeline into a building block that developers can use to deploy pipelines in either stand-alone or in a distributed manner as this image enable them to create clones by using the Gearbox capsule.

The manager, Sync(s) and Gear(s) have designed to be placed at the same Gearbox Capsule as they share the resources of the virtual containers. Nevertheless, a Gearbox Capsule could either include a whole Gearbox (pipeline) or only one gear (filter) but in both cases each Gearbox Capsule will include at least a Sync and a Manager.

It is possible for developers to build a distributed software pipeline by using a set of Capsules deployed on a cluster (For instance Kubernetes or Docker swarm). The Manager and even Syncs can be removed from the Gearbox Capsule and use them in a separated manner to build large scale distributed systems but this type of scenario is not considered in the version of Gearbox presented in this paper.

3.5.2. Implementation details

The Docker platform was used to create the capsule virtual container image [25].

The middleware was developed in C and implemented in the form of an I/O library by using Inter Process Communication Shared Memory (a python version is under construction). It is expected to use this middleware in two ways: the first one is using it as a service by including the library in the applications that developers would include in a pipeline by setting the place of each application in a pipeline as well as the input and output interfaces of each application. This information enables the capsule to launch applications and chaining them in the form of a pipeline system. The second one is using it as a pipeline engine, which oversees the executing and chaining of pipeline applications. Each middleware includes a pipeline system configuration file with which it invokes and

launches, in automatic and transparent manners, all the instances required to create a Gearbox, which are deployed by the virtual container of a capsule on a given infrastructure.

The *Sync* components as well as and *Workload Dispatcher* instances also have been developed in C and have been implemented by using a client–server model.

4. Methodology: Experimental evaluation

In this section, we describe a methodology based on two case studies that were designed for evaluating the efficiency of software pipelines that were modeled as Gearboxes encapsulated into virtual containers; as a result, two prototypes based on the gearbox model were developed (one per case study).

In the first case study, the software pipelines were evaluated on-premise and the data processing and results delivery have been measured locally (stand-alone). A prototype of the preprocessing system was developed to conduct this study. The prototype processes contents in three steps: (i) it reads the contents from a data source (a HD partition) where two types of content repositories (PDFs and medical images) were stored in, (ii) it executes three types of pipeline systems (pipelines including 2, 3, and 4 filters), and (iii) it delivers the results to a data sink (HD partitions).

Table 1
IT infrastructure features.

Cloud instances	OS	Cores	Processor type	Mem	HD
Client1	Centos7	12	Intel(R) Xeon(R) CPU E5–2640	64 Gb	3 Tb
Client2	Centos7	16	Intel(R) Xeon(R) CPU E5-2650 v2	64 Gb	3 Tb
CloudService	Centos7	4	Intel(R) Xeon(R) CPU E5-2603 v2	32 Gb	6 × 2 Tb

In the second case study, a version of the stand-alone prototype was incorporated into a cloud storage service in the form of a pre-processing service for securing data in-house before sending them to a private cloud. In this prototype, the medical image repository has been stored at the data source, whereas the data sink has been placed at a cluster of virtual containers built with Docker Swarm and providing a private cloud storage services built with Open Stack.

Table 1 shows the infrastructure where the prototypes where deployed on.

4.1. Repositories

Two types of repositories were considered in this experimental evaluation. The first one has been a PDF repository stored in the content delivery system of our research group (50 thousand documents). The second repository is the Digital Database for Screening Mammography called *DDSM* [26], which consists of 11 thousand images.

A bot software instance created three subsets of files from the two repositories in a random manner. The subsets contained between 500 and 15,000 elements (PDF: PDF5K, PDF10K and PDF15K; DDSM: DDSM0.5K, DDSM1K and DDSM1.5K). The bot created folders (one per subset of contents) in the data source and sent the contents to the corresponding target folders, from where the pipeline systems extracted contents to process them. Table 2 shows the characteristics of the subsets used in the experimental evaluation.

4.1.1. Metrics

In order to measure the performance of the capsules and software pipeline systems evaluated in this paper, we defined the following metrics.

- **Processing Time:** This is the time required for a pipeline system to process a subset of files. In the case of a Gearbox capsule, this time also includes the time spent by the collaborative system to manage the workload with the gearbox model calculated.
- **Percentage performance gain:** *gTR* represents the percentage of improvement in the response time (*tR*) of the Gearbox capsules in comparison with the evaluated solutions. $tR_{s1} - tR_{s2}$

$$gTR = \frac{tR_{s1}}{tR_{s2}} \cdot 100 \quad (7)$$

Where $s1$ and $s2$ represent the response time of solutions in each comparison.

4.1.2. Gears implemented to build software pipelines

Five gears were developed to create solutions based on software pipelines.

- **LZ4:** This gear implements a traditional data compression algorithm [14]. This gear is used to reduce the size of the contents $|C|$ to improve the cloud storage utilization. This gear transforms an incoming content $|C|$ into a new version $|C_c|$.
- **AES:** This gear implements the traditional algorithm AES [27] developed by OpenSSL to encrypt/decrypt incoming content $|C|$ or $|\hat{}C|$ by using keys of 128, 192, and 256 bits length to encrypt data.
- **IDA:** This gear implements a dispersal information algorithm [28], which receives a content $|C|$ that is split into n redundant data portions called dispersal Ds of size $L = |C|/m$ where m represents the number of Ds sufficient to recover $|C|$. It is required that $m < n$. This means a $|C|$ can be recovered even when $n - m$ of Ds are lost. The costs of this fault tolerance technique can be calculated in terms of extra capacity $EC = (n/m) \cdot |C|$.
  This algorithm can be configured by changing the parameters of n and m. In the *IDA* gear used in the experiments, these parameters were fixed to $n = 5$ and $m = 3$. This means the gear receives a file $|C|$ and delivers five Ds, $|C| = |d_1|,|d_n|..|d_5|$ [11]. The extra capacity produced by this gear is 66.7%, which enables the cloud storage to withstand the failure of 2 storage locations.
- **SHA3-256**: This gear calculates a Hash for each received content $|C|$. It produces a file H, which is used by end-users to verify the integrity of $|C|$.
- **CURL**: This gear sends all incoming contents to a cloud storage service by producing streaming through HTTP.

5. Experiments and results

In this section, we describe the results of the case studies conducted in the experimental evaluation.

5.1. Case study: Stand-alone

In this section, we analyze the results of evaluating the solutions/configurations when using subsets of PDFs and DDSM repositories.

5.1.1. Gearbox configurations and studied solutions The following software pipelines were studied:

- **Sacbe** [18]**:** This solution includes the chaining of filters by using pipes of the operating system. In order to perform a fair comparison, the data flow in the software pipelines was performed by using memory pipes of the operating system added to a virtual container. This approach represents an available solution to create pipe and filter patterns in an efficient manner for real-world scenarios.
- **Parsl** [17]**:** This solution includes the chaining of filters by using an engine for executing data-oriented workflows (dataflows). The dataflows of this engine produce task parallelism in the very fashion performed by the solution presented in this paper. In order to perform a fair comparison, Parsl was configured for creating data flows in software pipelines by using as many threads as physical cores in the servers where all the solutions were tested.
- **Gearbox:** This solution is created by chaining filters by using the gearbox model. Configurations of gearbox were also evaluated to observe the impact on performance of each component of the collaborative system considered in this solution.

Table 2
Subsets of contents used in the experimental evaluation.

		PDFs			DDSM		
Total	Amount	5k	10k	15k	0.5k	1k	1.5k
	Size	5.135 GB	10.514 GB	16.296 GB	13.419 GB	26.878 GB	40.204 GB
Per unit	Average	1.03 MB	1.05 MB	1.14 MB	26.83 MB	26.87 MB	27.11 MB
	Median	0.43 MB	0.44 MB	0.43 MB	15.46 MB	25.46 MB	25.77 MB

Table 3
Qualitative comparison of Gearbox with State-of-the-art solutions.

	Continuity			Adaptability		Efficiency		Reliability	
	Delivery	Integration	Runtime	Configuration			In-memory storage	Parallelism	Fault tolerance
MakeFlow				✓					
OpenShift				✓		✓		[a]	[b]
Jenkins				✓		✓	✓		[b]
Parsl				✓		✓	✓		[c]
Sacbe				✓	✓		✓		
Gearbox				✓	✓	✓	✓		[c]

[a]Means this solution is using Jenkins. [b]Involves by using Kuberneters. [c]Means that the solutions offer fault tolerance only at pipeline level (not solving failure at stage level).

- **Gear-Sync:** This is a gearbox configuration that is not considering the cloning mechanism, but only the workload management performed by the Syncs and middleware of each gear.
- **Gear-P:** This is a version of *Gear-Sync* configuration but enabling the cloning of filters. In this configuration no adjustments are performed in the gearbox layout in execution time. This means the first configuration defined by the workload manager is used to process all the contents in a data source.
- **Gearbox-CS:** This is a version of *Gear-P* but the manager performs adjustments, in execution time, in the gearbox configurations in an elastic manner.

Several pipeline software systems are currently available (e.g. Parsl, Sacbe, MakeFlow, OpenShift or even Jenkins) for designers to build software pipelines. We have chosen Parsl and Sacbe to conduct the experimental evaluation and the case studies because both solutions are very close to offer the properties offered by Gearbox. For instance, both solutions build software pipelines by using parallelism in the case of Parsl, and in-memory storage in the case of Sacbe. See more details about these and other solutions of the state-of-the-art in a qualitative comparison showed in Table 3 where are analyzed properties such as adaptability, efficiency and reliability. This comparison is described in the related work section.

The solutions and configurations were modified by adding gradually filters to a pipeline from 2 to 3 and to 4 filters. Each variation in the configuration was evaluated by using the repositories (PDF: PDF5K, PDF10K and PDF15K; DDSM: DDSM0.5K, DDSM1K and DDSM1.5K).

5.2. Preprocessing of medical image repository (DDSM)

Fig. 5 shows different results for response times (vertical axis) produced by the studied solutions for different number of filters when processing subsets of contents extracted from the DDSM repository (horizontal axis). Fig. 5a shows response times for two filters; Fig. 5b for three filters and Fig. 5c for four filters.

Fig. 5 also shows results of the performance gain between the pipeline systems implemented by using gearbox capsules with regular pipeline systems when using different numbers of filters. Fig. 5d shows this metric for two filters; Fig. 5e for three filters and Fig. 5f for four filters.

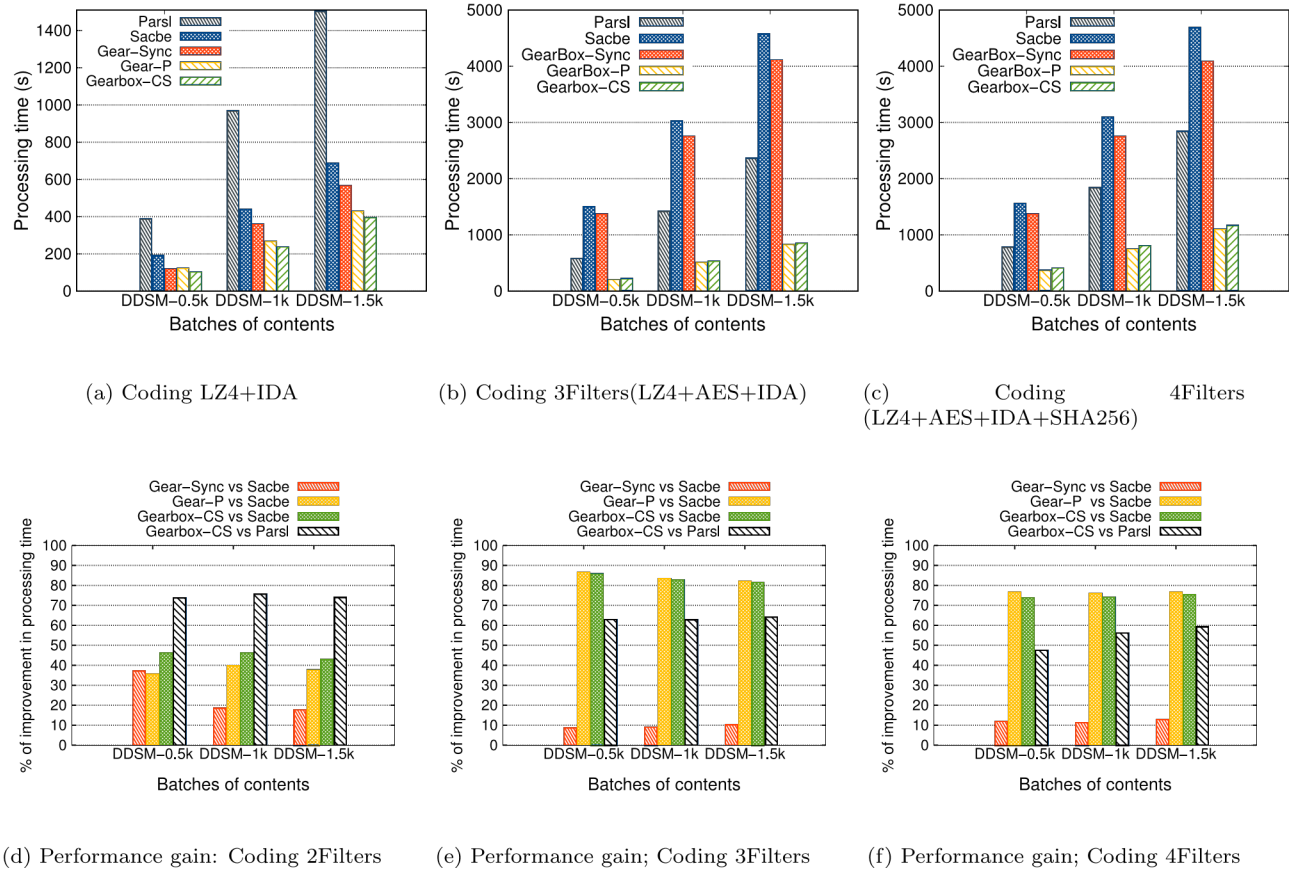
As it can be seen, the synchronization of the input/output data of all filters in a pipeline is good enough for a software pipeline (Gear-Sync) using in-memory storage service to produce better performance than another in-memory pipeline (Sacbe) even when increasing the number of filters in the pipelines (see Figs. 5a, 5b and 5c).

Gear-Sync produces up to 36% better response times than Sacbe with few filters (see Fig. 5d), which is reduced to 10% when increasing the number of contents to be processed (see Figs. 5e and 5f).

As expected, Parsl ' ' does not offer a ' competitive performance for large files (see Fig. 5a) as this type of engine has been designed for producing task parallelism, which is more suitable for processing small contents. This behavior was repeated for the pipeline including three and four filters.

We decided to study the poor performance of Parsl when processing big size files. We found that, when Parsl launching threads in a server of twelve cores (one thread per core available in the server used in this experiment), a queuing of the file read operations was observed. This queuing kept the filers of the pipeline idle for a period proportional to the service time spent in these read operations. To achieve a competitive configuration of Parsl, we performed a set of experiments by varying the number of threads used by Parsl until finding a configuration that would reduce that queuing of read and write operations, which enabling the filters to start the processing of data in less time. We found that reducing the number of threads to the half (6) was key for Parsl to read contents in overlapped manner with the threads used by the filters for processing data in the pipeline. This reduced the response time of Parsl in significant manner (see Figs. 5a–5c).

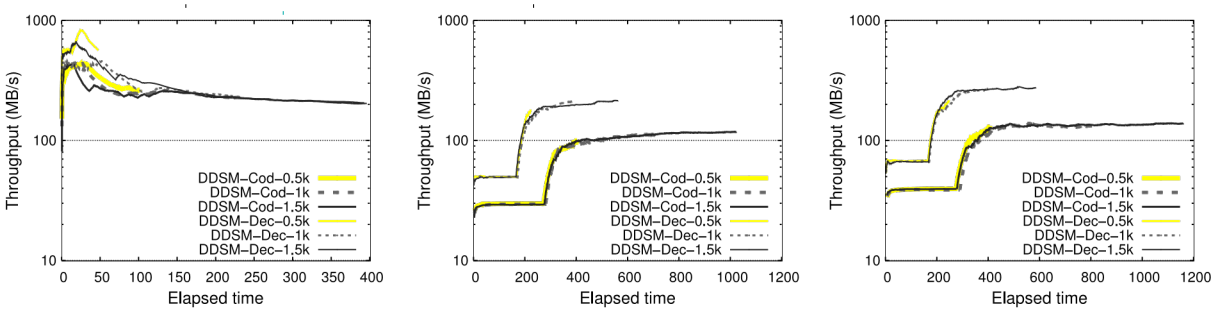
In the first gearbox layout that includes gear up/down (*Gear − P*), the improvement of response times is significant even when increasing the number of filters (see Gear-P in Figs. 5a–5c). It seems that the performance is not significantly affected when increasing the volumes of contents, which is observed in Parsl configurations (from 47,45% to 75,5%). In fact, the percentage of performance gain of Gear-P is up to 40% better than Sacbe when deploying few filters (See Fig. 5d), which is increased to 88% when increasing to three filters to reach 75% when using four filters. When increasing the number of contents to be processed, the improvement produces a fluctuation of ±10% (see Figs. 5d, 5e and 5f). The drawback of the Gear-P configuration is that the number of nodes must be fixed at configuration time by knowing the *RT*



(a) Coding LZ4+IDA (b) Coding 3Filters(LZ4+AES+IDA) (c) Coding 4Filters (LZ4+AES+IDA+SHA256)

(d) Performance gain: Coding 2Filters (e) Performance gain; Coding 3Filters (f) Performance gain; Coding 4Filters

Fig. 5. Processing and response time of software pipelines when coding DDSM repositories by using 2,3 and 4 filters.



(a) Throughput per filter coding/decoding (LZ4+IDA)

(b) Throughput per filter coding/decoding (LZ4+AES+IDA)

(c) Throughput per filter coding/decoding (SHA256+LZ4+AES+IDA)

Fig. 6. Coding/decoding throughput and performance difference per filter when processing DDSM repositories.

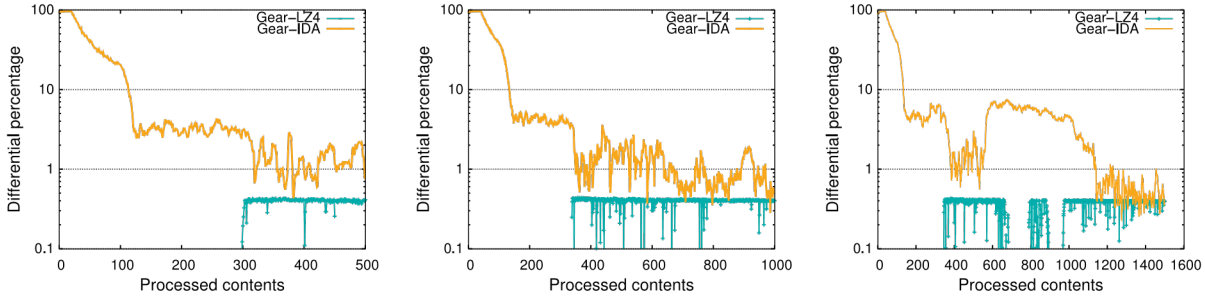
and *DT* of the gears in advance, and it cannot be changed during execution time.

In turn, the Gearbox-CS configuration modifies the gearbox layout during execution time depending on the workload[3] and this enables the Gearbox capsule to overcome bottlenecks produced during workload processing. This configuration does not only produce a better performance gain than Gear-P (Min = 8,3 and Max = 16,3), Sacbe (Min = 42.75%, Max = 85.63% and Parsl (Min = 47.45% and Max = 75.51%), but Gearbox-CS can also handle changes in the workload or request accumulation within the input queues. Moreover, this configuration offers better performance than Gear-P when contents producing queuing (see Gear-P and Gearbox-CS in Fig. 5d for the different subset of contents). When moving the size of the bucket's limit for each pair of gears in the Gearbox-CS configuration, a performance improvement of Gearbox-CS of up 40% is also observed in comparison with Gearbox-P.

The decoding software pipelines produced a similar behavior to that produced by the coding software pipelines and reduced the response times in a range of (34-91]%.

To understand how Gearbox-CS manages the workload during execution time to homogenize the retrieval/delivery throughput rates of the gears, we focus our attention on the throughput produced by each gear in the pipeline.

Fig. 6 shows the throughput (vertical axis) produced by the configuration (Gearbox-CS) during both the coding and decoding of subsets of contents in the DDSM repository.



(a) Coding of the 0,5K DDSM subset

(b) Coding of the 1K DDSM subset.

(c) Coding of the 1.5K DDSM subset.

Fig. 7. Coding/decoding performance difference per filter when processing DDSM repositories.

As it can be seen, Gearbox homogenizes the gears' throughput, in execution time, through three phases.

In the first one, the bottlenecks are controlled (see how in very early in Figs. 7a, 7b and 7c the throughput is increased) by performing gear up/down.

In the second phase, gearbox performs adjustments to increase the throughput of driven gears (slow applications), which improves the global throughout (see first part in Fig. 6a where a throughput variation is even better than a homogenized throughput). However, this increment in the throughput produces a queuing in the driver gears (fast ones) and converts them into driven, which reduces the throughput (downward-sloping lines shown in Fig. 6a). The homogenization algorithm stops this throughput reduction when the throughput variation of all gears is controlled. When more gears are included in the pipeline, the homogenization of the gears' throughput produces an exponential improvement (see Figs. 6b and 6c).

In the last phase, the behave of the pipeline throughput is stabilized and few changes are performed by gearbox to keep the system fine-tuned. Fig. 7 also shows that the software pipelines for decoding contents produced a similar behavior to that produced by the coding software pipelines but the stabilization was early achieved for these pipelines.

[3] The manager also can absorb the changes in the infrastructure but this is not considered in this paper.

Fig. 7 shows the ' ' difference of the ' service times (vertical axis) between the gears LZ4 and IDA with the fastest gear (AES). This difference was captured when the gears, included in a software pipeline, coding contents of the 0.5k, 1k and 1.5k subsets (horizontal axis) included in the DDSM repository (see Figs. 7a, 7b and 7c).

Fig. 7 again allows us to observe the homogenization and balancing procedures performed by the gearbox model. As it can be seen, the IDA gear produced the slower performance, which producing the highest performance difference compared to the fastest gear (AES). This behave (more than 100% slow-down in comparison with the performance of AES gear) is produced because of the expensive and extensive I/O and computing operations performed in the IDA gear. We also can observe that Gearbox gradually reduces this difference among all the filters until it is stabilized in a range of (10%, 0.1%].

It can also be observed that the more contents to be processed by a pipeline, the better it is possible to balance the gear performance. The difference of response times among the filters was less than 5% for the 0.5k subset (see Fig. 7a), 1% for the subset 1k (see Fig. 7b) and less than 1% for subset 1.5k (see Fig. 7c). The process of content decoding produces a smooth behave similar to coding process. In these cases, the stabilization was early achieved compared to the coding procedures as less information is processed and produced in this process than in the coding procedure.

As it can be seen, gearbox converts a pipeline that includes applications producing heterogeneous performance into a system with homogeneous retrieval/delivery throughput; as a result, a nearly continuous dataflow is produced through the software pipeline, which enables the gearbox model to reduce bottlenecks and idle times in a significant manner. This shows the efficacy of the gear up/down strategies when performed during execution.

5.3. Processing small files from a PDF repository

Fig. 8 shows the response times (vertical axis) produced by the studied solutions for different numbers of gears when processing subsets from the PDF repository (horizontal axis). Fig. 8a shows results for two filters, Fig. 8b for three and Fig. 8c for four filters.

As it can be seen, the behavior of the software pipelines when processing subsets from the PDF repository is quite similar to that observed when the software pipelines process subsets from the DDSM repository. As it has been expected, *Parsl* becomes competitive for this scenario when executing two gears and even better than the in-memory solution (Sacbe) for 15k-Two-Gears as well as for scenarios where the pipeline includes three or four gears.

In this scenario, the improvement of Parsl in comparison with its performance when processing DDSM has been due to the task parallelism produced by the threads invoked by this engine as well as the small sizes of the contents that are processed in this scenario. This engine enables Parsl to assign more workload to the cores where the applications are executed, which reduces the idle periods of the applications and increases the utilization of the resources.

Gearbox-CS produces the best performance compared with the Sacbe engine and even with other configurations of gearbox. The configurations also produce better results than *Parsl* when this solution used all available cores in the server. On average, for a system of two stages, gearbox yielded a performance gain of 58.36% in comparison with Parsl (64.31% for 5k, 58.36% for 10k and 46.22% for 15k). In the case of the three stages pipeline system, the performance gain was 43.09% (62.26% for 5k, 40.30% for 10k and 26.72% for 15k), whereas for a pipeline system of four stages the gain was 49.65% (53.78% for 5k, 53.78% for 10k and 46.19% for 15k).

As it can be seen, pipelines deployed in the form of gearbox *capsules* can not only improve the performance compared to regular software pipelines but it is also possible to easily add more properties to the contents by spending even less costs than a regular pipeline created by other engines (e.g., gearbox spent 461.97 s for adding 4 properties to the contents, whereas Parsl spent 675 s to add only two value properties).

As expected, the efficiency of a pipeline system built with gearbox depends on the resources available in the virtual container where the Gearbox capsule is deployed on. The main resource properties are the number of cores (required by the clones created when performing gear up procedures) and the

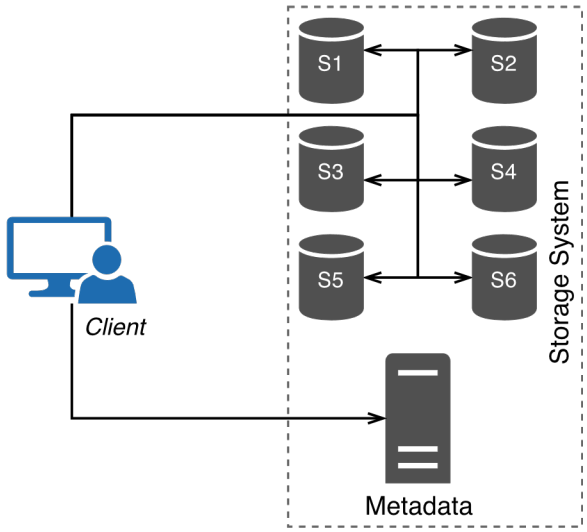
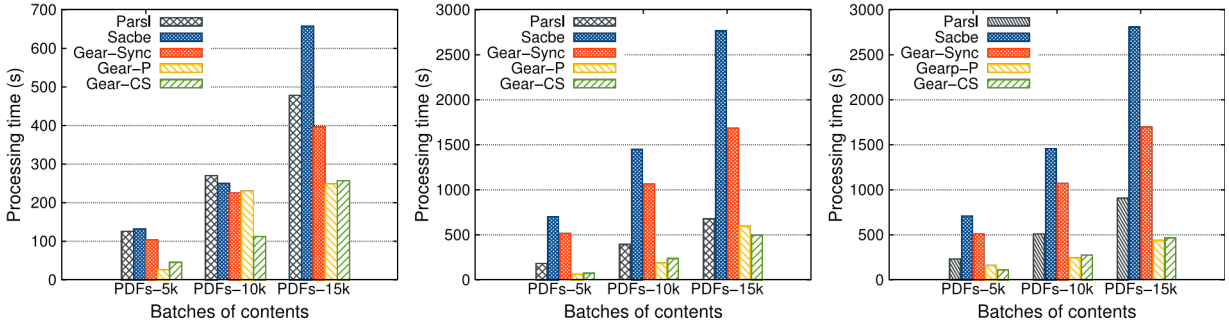


Fig. 9. Private cloud storage service.



(a) Coding LZ4+IDA (b) Coding 3Filters(LZ4+AES+IDA) (c) Coding 4Filters(LZ4+AES+IDA+SHA256)

Fig. 8. Processing small files (PDFs) by using software pipelines.

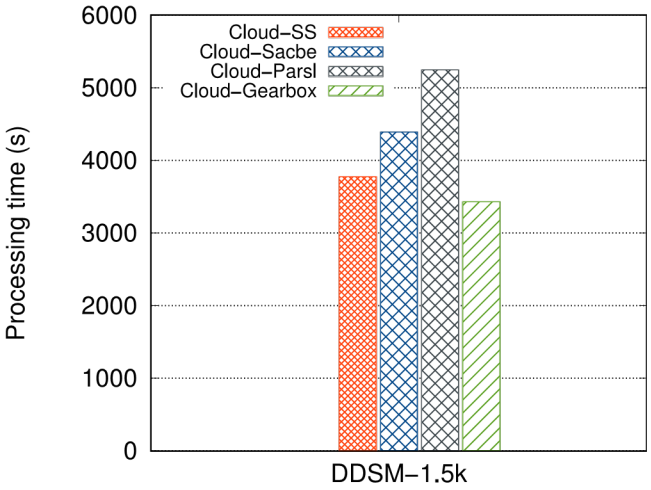


Fig. 10. Coding response time when processing 1.5k subset from DDMS

memory (required by clones when performing gear down procedures). Gearbox therefore performs gear up procedures to balance the throughput (the best scenario) and gear down is used to reduce the differences that could not be solved by the gear up procedure because of infrastructure limitations.

6. Configuration and results of a study case based on the incorporation of a capsule into a cloud storage service

The previous experiments have shown that gearbox represents an efficient and flexible method for building pipelines. The gearbox properties can also be used by cloud providers to add properties to the contents of their end-users/customers. In this section, we evaluate end-to-end solutions that enable end-users to add in-house properties to their contents, which avoids cloud providers to perform this type of task.

This type of solution requires end-users to execute an in-house preprocessing tool to prepare their contents before sending them to the Cloud infrastructure defined by the organization for the content preservation (e.g., a public/private/hybrid cloud). These solutions are quite useful in the cases where end-users are not associated to a given organization. The following solutions were implemented to manage this study case:

repository.

- **Cloud-SS:** This solution only used the filter CURL to send a set of files to a private cloud storage service. This solution represents a common service for end-users but without adding properties to their contents. It is expected that this configuration would produce the lowest response times for end-users.
- **Cloud-Gearbox:** This solution incorporates a *Gearbox-CS* capsule including four filters (SHA, AES, LZ4 and CURL) to a cloud storage service. This solution represents the implementation of gearbox as an end-to-end preprocessing tool for a cloud storage service.
- **Cloud-Sacbe:** This solution incorporates a *Sacbe* configuration including the very four filters in a cloud storage service. This solution is expected to reflect the overhead of adding properties to the contents.
- **Cloud-Parsl:** This solution incorporates a *Parsl* configuration including the filters in a cloud storage service. This solution is expected to reflect the overhead of adding properties to the contents.

The goal here is to detect the performance overhead created by adding a software pipeline to the clients of cloud storage services. The solutions in this study case were evaluated by processing a subset of contents (DDSM1.5K) in a private cloud storage service.

Fig. 11. Breakdown of response time used.

Fig. 9 shows a private cloud storage service [11] including a client, which implements both the Gearbox capsules and the preprocessing solutions considered in this case study.

Fig. 10 shows the response time for the configurations evaluated in this study case. As expected, a regular software pipeline in including *SHA, AES, LZ4 and CURL* (optimized by using in-memory dataflows) produced an overhead of 13.95%, whereas Cloud-Parsl produced an overhead of 28.02%

In turn, the client using the gearbox model does not produce overheads but also improves the user experience by 12.5% in comparison with the configuration where the end-user does not add properties to her/his contents. When removing the cloud storage service latency, the performance gain of gearbox in comparison with Sacbe and Parsl is the same that has been shown when evaluating software pipelines in the stand-alone case studies.

In addition, the management of operations required by the gearbox implementation are not affecting the filters' performance in a significant manner and this allows the model to improve overall throughput of the software pipelines in a significant manner.

Fig. 11 shows the Breakdown of response time of Gearbox managing a four filter pipeline considered in the case study, which represents the sum of the service times of the gears (G_1, G_2, G_3, G_4). The costs of the communication overhead generated for the sake of handling different data processing and management tasks in Gearbox are showed in Fig. 11 by *Management* and *ShMem*(*R/W*) (Shared Memory service).

Management time includes tasks such as initiation of the collaboration system, scanning data in source path, creating syncs for Gears and creating messages for shared resource patterns. This time represents 0.001% of the response time. *ShMem*(*R/W*) represents the read and write operation costs within Gearbox capsule. This time represents the 2.36% of the response time.

The read of files (Read Files) is assumed by Gear 1 or *G*1 and represents the cost of reading contents from the data source, whereas *Transfer data* represents the delivery of processed contents to the private cloud storage service. This time also represents the service time of Gear 4.

7. Related work

Several pipeline software systems are currently available (e.g. Parsl [29], Sacbe [18], MakeFlow [30], and OpenShift [31] or even Jenkins [32]) for designers to build software pipelines. We performed a qualitative comparison of Gearbox with other solutions from the state-of-the-art for analyzing properties such as continuity, adaptability and reliability, which are showed in Table 3. This table will be used to describe the differences between Gearbox model and solutions available in literature.

In this study of the state-of-the-art were identified two types of solution similar to Gearbox. The first one refers to solutions directly focused on the building of software pipelines to achieve continuous integration and continuous delivery (CI/CD widely used in industrial scenarios such as MakeFlow [30], OpenShift [31] and Jenkis [32]). The goal of this type of solution is the integration and the delivery of software to production. The efficiency of the processing of data in these pipelines is not a priority for these solutions.

The second one refers to solutions focused on not only in the building of software pipelines but also in the creation of dataflows, which is feasible when the CD/CI is granted (Parsl Sacbe [18], and even in some homemade solutions built by using Jenkins).

A *data flow* architecture processes data with a structure in form of graphs. The vertices of the graph represent asynchronous tasks, although they are often only instructions, the edges of the graph represent communication routes that carry the execution

results needed as operands in subsequent instructions [33]. The solution proposed in this research work is based on the usage of the architectural pattern of data flows, for which the prototype of the solution and comparison was implemented from this type of pattern.

For dataflow solutions in software pipelines were identified both ad hoc and generic-centric solutions to create dataflows.

In the case of ad hoc solutions, Grawinkel et al. [20] proposed a special purpose solution to create pipes developed in the C++ programming language as an alternative to the implementation of Unix shell pipes. The implementation of the proposed pipes consisted in the processing of the data for integrity (calculating a hash value), reduction of storage space (using compression algorithms) and encryption to increase the confidentiality of data stored in Unix. The results showed an increase of performance in the pipes in C++, connecting their stages directly and working with multiple pipes, independent and in parallel. Although this approach easily outperformed the UNIX pipes used by command line tools, this solution is only useful for the filters considered in this study, whereas both Gearbox and the other solutions studied in this paper enable developers to create different types of software pipelines.

In a similar way, Folarin et al. [34] presented NGSeasy, a system in which containerization and pipeline execution techniques were integrated with focus on workflows for Next Generation Sequencing (NGS) by using a set of modular and reusable building blocks based on virtual containers that can be versioned. NGSeasy is composed of a method to be implemented in scenarios where the data cannot be shared (by size or for privacy reasons) and the analysis must be done on-site. In this context Dayal et al. [35] described an approach that addresses challenges that arise in online data analysis, including how to efficiently run multiple analysis components in the test area resources. This type of solution is mainly focused on specific data analytic, where the creation of multiple pipelines is not immediate, which is the case of the solutions studied in this paper.

In past were proposed solutions focused on the building of workflows on peer-to-peer (P2P) and the Grid. This is the case of a java-based solution called Triana that was created by Taylor et al. [36] to support services within multiple environments, integrating several types of middleware tools. Triana builds workflows with multiple services and interfaces to creating processing pipes through the union of tools and services. This type of workflow-based solution is not considered in this paper as continuous dataflow is not the goal of this type of solutions but also the continuous integration and delivery.

In the case of generic-centric software pipelines, that is the case of *Sacbe* [18], *Parsl* [29] were found offering similar properties to Gearbox. Sacbe, proposed by Yanes et al. is a modular software architecture for secure, reliable and flexible end-to-end cloud storage. This architecture is based on pipes designed as architectural software patterns to move data/metadata in continuous data flows (continuous flow input to the pipeline) from user devices to storage locations in the cloud, through modules of coverage treatment (e.g., confidentiality, reliability) or management (e.g., sharing, authentication). These coverage pipelines transport and process content from a data source (any of users' devices, hard drive partitions or shared folders) to cloud storage locations, in a continuous data stream. This type of pipe represents a virtual path that is called *Data Sacbe*. The interoperability of building blocks (BBs) in the pipes is facilitated by a set of I/O interfaces, with which each BB sends/receives data flows from/to another BB. Sacbe provides continuity, reliability based on replicated BBs, adaptability at configuration time and efficiency based on in-memory storage. Gearbox not only includes these properties but also it adapts its layout configurations to the workload by managing in-memory storage and yielding implicit parallelism.

The models based on building blocks improve the deployment of pipes in different environments (operating system, cloud, cluster, workstation) for development, testing and production [37]. In these models the pipes are encapsulated in virtualized software structures called containers [37], which contain the necessary software to run the application that is being encapsulated. This method of encapsulation provides efficiency, portability, version control, and reproducibility, as well as flexibility in the deployment of solutions allowing containers to be created or destroyed quickly [38]. This building model was also incorporated to Gearbox by the usage of Gearbox Capsule images, which are basically a building block.

Parsl is a library created for building both workflows and software pipelines with special focus on producing implicit dataflows. This solution enables developers to execute applications concurrently while respecting data dependencies. Parsl solution produces pipelines by executing code or applications by following a code sequence and produces task parallelism by using threads for each stage, which are defined at configuration time. In turn, Gearbox, produces the implicit parallel patterns on-demand, considers synchronization of the applications I/O operations and in-memory storage for not only producing dataflow but also a constant and quasi-continuous data processing flow. It is feasible for the solutions studied in this paper (both Sacbe and Parsl as well as other similar to those) to take advantage of efficiency and adaptability properties of Gearbox model to reach a dataflow as continuous as possible. In similar way, Gearbox model can be improved by incorporating more properties such as reliability and resilience for dataflows as well as to produce geographically distributed solutions that are already considered by current solutions available in the state-of-the-art. We consider Gearbox can be a complement, for instance, for function as a service as OpenShift to build in-memory, workload adaptable and parallel software pipelines. The Gearbox Capsule produces a deployment configuration file that can be sent to Kubernetes [39] for deployment of the pipeline solutions in a similar way OpenShift uses Jenkins to do this task.

8. Conclusions

This paper presented a container-based processing model for application pipelines to process large volumes of data in an efficient manner.

This model is based on the gearbox metaphor where gears represent applications, whereas boxes represent pipeline systems. In the gearbox model, a workload manager based on a collaborative system automatically performs gearing up (by using parallel patterns) and/or gearing down (by using in-memory storage) until applications in the pipeline produce a uniform data retrieval/delivery throughput. The manager was developed as a deployment software piece called *capsule* encapsulated into virtual containers. This virtual container also included the pipeline systems built by organizations and modeled as a gearbox. These capsules were evaluated and compared with traditional services, through experiments processing data from repositories of medical images and PDFs. A study case was also conducted based on an implementation of a pre-processing pipeline system in the form of a capsule, which was incorporated to a cloud containerized storage service for supporting medical imagery repository management.

The experimental evaluation revealed that the gearbox model reduces delays and bottlenecks produced by the heterogeneous performance of the applications included in the pipeline system. This significantly improves the end-user service experience when pre-processing large-scale data in comparison with available pipeline solutions. In comparison with an in-memory pipeline system, the improvement of response times observed in all experiments was in the range of (43%, 91%] for different numbers of applications in the pipeline systems, for the different repositories processed and the different studies conducted. When a pipeline included more than three applications in the system, the performance improvement was in a range of (81%, 86%) and when increasing the volumes of data to be processed, the improvement range was (89%, 91%]. This means organizations can add control properties to large data volumes not only without suffering overheads but also improving the performance of cloud storage services. In comparison with engines producing parallel workflows as Parsl, gearbox even increases the percentage of the performance gain when processing large contents, whereas gearbox preserves 58.33% of performance gain on average, which is in a range of [64.31%, 26.72%].

In the comparison with a traditional cloud storage, the evaluation revealed that when incorporating a *capsule* to a cloud containerized storage service, the end-users can ensure their contents in-house before sending them to preservation (e.g. a public/private/hybrid cloud).

The capsule based on gearbox did not produce overheads but improved the user experience by 12.5% in comparison with a configuration where the end-user does not add properties to her/his contents. When removing the cloud storage service latency, the performance gain of gearbox in comparison with regular pipeline is the same that showed when evaluating software pipelines for two filters. These results show the feasibility of applying this model, for instance, to pipe&filter systems in diverse real-world scenarios.

We are currently working on inexpensive control algorithms based on fussy logic algorithms for the Manager of the capsule to improve the load-balancing procedure during execution time. We also are exploring scheduling algorithms for creating gearbox layouts in advance for static infrastructures.

Acknowledgment

This work has been partially supported by the ‘‘Spanish Ministerio de Economia y Competitividad’’ under the project grant TIN2016-79637-P ‘‘Towards Unification of HPC and Big Data paradigms’’.

References

[1] J. Gantz, D. Reinsel, The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east, in: IDC IView: IDC Analyze the Future 2007, 2012, pp. 1–16.

[2] Rachel Dines, The state of disaster recovery preparedness, Disaster Recov. J. 24 (1) (2011) 1–3.

[3] J. Tsidulko, The 10 biggest cloud outages of 2015 (so far), 2015, https://www.crn.com/news/cloud/index/cloud-outages-cloud-servicesdowntime.htm.

[4] R.H. Sloan, R. Warner, Unauthorized Access: The Crisis in Online Privacy and Security, CRC press, 2013.

[5] G. Laatikainen, O. Mazhelis, P. Tyrväinen, Role of acquisition intervals in private and public cloud storage costs, Decis. Support Syst. 57 (2014) 320–330.

[6] J.L. Gonzalez-Compean, Oscar Telles, Ivan Lopez-Arevalo, Miguel MoralesSandoval, Victor J. Sosa-Sosa, Jesus Carretero, A policy-based containerized filter for secure information sharing in organizational environments, Future Gener. Comput. Syst. 95 (2019) 430–444.

[7] C. Lee, C.C. Lee, S. Kim, Understanding information security stress: Focusing on the type of information security compliance activity, Comput. Secur. 59 (2016) 60–70.

[8] J. Bethencourt, A. Sahai, B. Waters, Ciphertext-policy attribute-based encryption, in: 2007 IEEE Symposium on Security and Privacy (SP ’07), IEEE, 2007, pp. 321–334.

[9] Praveen Kumar, P.J.A. Alphonse, et al., Attribute based encryption in cloud computing: A survey, gap analysis, and future directions, J. Netw. Comput. Appl. 108 (2018) 37–52.
[10] Y. Jiang, W. Susilo, Y. Mu, F. Guo, Ciphertext-policy attribute-based encryption against key-delegation abuse in fog computing, Future Gener. Comput. Syst. 78 (2018) 720–729, http://dx.doi.org/10.1016/j.future.2017.01.026.
[11] J.L. Gonzalez, J.C. Perez, V.J. Sosa-Sosa, L.M. Sanchez, B. Bergua, Skycds: A resilient content delivery service based on diversified cloud storage, Simul. Model. Pract. Theory 54 (2015) 64–85.
[12] J. Spillner, J. MüLler, A. Schill, Creating optimal cloud storage systems, Future Gener. Comput. Syst. 29 (4) (2013) 1062–1072.
[13] S. Zhang, W. Wang, H. Wu, A.V. Vasilakos, P. Liu, Towards transparent and distributed workload management for large scale web servers, Future Gener. Comput. Syst. 29 (4) (2013) 913–925.
[14] S.-J. Kwon, S.-H. Kim, H.-J. Kim, J.-S. Kim, Lz4m: A fast compression algorithm for in-memory data, in: International Conference on Consumer Electronics (ICCE), IEEE, 2017, pp. 420–423.
[15] J.L. Ortega-Arjona, Patterns for Parallel Software Design, Vol. 21, John Wiley & Sons, 2010.
[16] F. Buschmann, K. Henney, D.C. Schmidt, Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing, Vol. 4, John Wiley & Sons, 2007.
[17] Y. Babuji, K. Chard, I. Foster, D.S. Katz, M. Wilde, A. Woodard, J. Wozniak, Parsl: Scalable parallel scripting in python, in: 10th International Workshop on Science Gateways (IWSG 2018), 2018.
[18] J.L. Gonzalez-Compean, Victor Sosa-Sosa, Arturo Diaz-Perez, Jesus Carretero, Jedidiah Yanez-Sierra, Sacbe: A modular software architecture for secure, reliable and flexible end-to-end cloud storage, J. Syst. Softw. 135 (2018) 143–156.
[19] J. Gonzalez-Compean, V.J. Sosa-Sosa, A. Diaz-Perez, J. Carretero, R. Marcelin-Jimenez, Fedids: a federated cloud storage architecture and satellite image delivery service for building dependable geospatial platforms, Int. J. Digit. Earth (2017) 1–22.
[20] M. Grawinkel, M. Mardaus, T. Süß, A. Brinkmann, Evaluation of a hash-compress-encrypt pipeline for storage system applications, in: International Conference on Networking, Architecture and Storage (NAS), IEEE, 2015, pp. 355–356.
[21] D.M. Ritchie, The evolution of the unix time-sharing system, in: Language Design and Programming Methodology, Springer, 1980, pp. 25–35.
[22] Robert A. Weber, Kevin M. O'Connell, Alternative futures: United States commercial satellite imagery in 2020, Department of Commercial, Commercial Remote Sensing Regulatory Affairs (2011).
[23] Accountability Act, Health insurance portability and accountability act of 1996, Public law 104 (1996) 191.
[24] M.A. Núñez-Gaona, R. Marcelín-Jiménez, J. Gutiérrez-Martínez, H. AguirreMeneses, J.L. Gonzalez-Compean, A dependable massive storage service for medical imaging, J. Digit. Imaging (2018) 1–12.
[25] C. Boettiger, An introduction to docker for reproducible research, Oper. Syst. Rev. 49 (1) (2015) 71–79.
[26] M. Heath, K. Bowyer, D. Kopans, R. Moore, W.P. Kegelmeyer, The digital database for screening mammography, in: Proceedings of the 5th International Workshop on Digital Mammography, Medical Physics Publishing, 2000, pp. 212–218.
[27] N.F. Pub, 197: Advanced encryption standard (aes), Federal Inf. Process. Stand. Publ. 197 (441) (2001).
[28] M.O. Rabin, Efficient dispersal of information for security, load balancing, and fault tolerance, J. ACM 36 (2) (1989) 335–348, http://dx.doi.org/10. 1145/62044.62050.
[29] Y. Babuji, A. Woodard, Z. Li, D.S. Katz, B. Clifford, R. Kumar, ..., M. Wilde, Parsl: Pervasive parallel programming in python, in: Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing, ACM, 2019, pp. 25–36.
[30] M. Albrecht, P. Donnelly, P. Bui, D. Thain, Makeflow: A portable abstraction for data intensive computing on clusters, clouds, and grids, in: Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies, ACM, 2012, p. 1.
[31] A. Lomov, OpenShift and Cloud Foundry PaaS: High-Level Overview of Features and Architectures, White paper, Altoros, 2014.
[32] J.F. Smart, Jenkins: The Definitive Guide: Continuous Integration for the Masses, O'Reilly Media, Inc, 2011.
[33] R. Duncan, A survey of parallel computer architectures, Computer 23 (2) (1990) 5–16.
[34] A.A. Folarin, R.J. Dobson, S.J. Newhouse, Ngseasy: a next generation sequencing pipeline in docker containers, F1000Research 4.
[35] J. Dayal, J. Cao, G. Eisenhauer, K. Schwan, M. Wolf, F. Zheng, H. Abbasi, S. Klasky, N. Podhorszki, J. Lofstead, I/o containers: Managing the data analytics and visualization pipelines of high end codes, in: Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), IEEE, 2013, pp. 2015–2024.
[36] I. Taylor, M. Shields, I. Wang, A. Harrison, The triana workflow environment: Architecture and applications, in: Workflows for E-Science, Springer, 2007, pp. 320–339.
[37] D. Bernstein, Containers and cloud: From lxc to docker to kubernetes, IEEE Cloud Comput. 1 (3) (2014) 81–84.
[38] S. Julian, M. Shuey, S. Cook, Containers in research: Initial experiences with lightweight infrastructure, in: Proceedings of the XSEDE16 Conference on Diversity, Big Data, and Science At Scale, ACM, 2016, p. 25.
[39] K. Hightower, B. Burns, J. Beda, Kubernetes: Up and Running: Dive Into the Future of Infrastructure, O'Reilly Media, Inc, 2017.