# Why Do Migrations Fail and What Can We Do about It?

Gong Zhang and Ling Liu
College of Computing, Georgia Institute of Technology, Atlanta, USA

## Abstract

*This paper investigates the main causes that make the application migration to Cloud complicated and error-prone through two case studies. We first discuss the typical configuration errors in each migration case study based on our error categorization model, which classifies the configuration errors into seven categories. Then we describe the common installation errors across both case studies. By analyzing operator errors in our case studies for migrating applications to cloud, we present the design of CloudMig, a semi-automated migration validation system with two unique characteristics. First, we develop a continual query (CQ) based configuration policy checking system, which facilitate operators to weave important configuration constraints into CQ-based policies and periodically run these policies to monitor the configuration changes and detect and alert the possible configuration constraints violations. Second, CloudMig combines the CQ based policy checking with the template based installation automation to help operators reduce the installation errors and increase the correctness assurance of application migration. Our experiments show that CloudMig can effectively detect a majority of the configuration errors in the migration process.*
**Keywords***:System management, Cloud Computing, Application Migration*
**Technical area***: Cloud Computing*

## 1 Introduction

Cloud computing infrastructures, such as Amazon EC2 [3], provide elastic, economical and scalable solutions and outsourcing opportunities for different types of consumers and end-users. Its pay-as-you-go utility-based computing model attracts many enterprises to build their information technology services and applications on the EC2-like cloud platform(s) and many successfully achieve their business objectives, such as SmugMug, Twistage and so forth. An increasing number of enterprises embrace Cloud computing by making their deployment plans or engaging in the process to migrate their services or applications from a local data center to the Cloud computing platform like EC2, because this will greatly reduce their infrastructure investments, simplify operations, and obtain better quality of information service.

However, the application migration process from the local data center to the Cloud environment turns out to be quite complicated: error-prone, time-consuming and costly. Even worse, the application may not work correctly after the sophisticated migration process. Existing approaches mainly complete this process in an ad-hoc manual manner and thus the chances of error are very high. Thus how to migrate the applications to the Cloud platform correctly and effectively poses a critical challenge for both the research community and the computing service industry.

In this paper, we investigate the factors and the causes that make the application migration process complicated and error-prone through two case studies, which migrate Hadoop distributed system and RUBiUS multi-tier Internet service from a local data center to Amazon EC2. We first discuss the typical configuration errors in each migration case study based on our error categorization model, which classifies the configuration errors into seven categories. Then we describe the common installation errors across both case studies. We illustrate each category of errors by examples through selecting a subset of the typical errors observed in our experiments. We also present the statistical results on the error distributions in each case study and across case studies. By analyzing operator errors in our case studies for migrating applications to cloud, we present the design of CloudMig, a semi-automated migration validation system that offers effective configuration management to simplify and facilitate the migration configuration process. The CloudMig system makes two unique contributions. First, we develop a continual query based configuration policy checking system, which facilitate operators to weave important configuration constraints into continual query policies and periodically run these policies to monitor the configuration changes

and detect and alert the possible configuration constraints violations. Second, CloudMig combines the continual query based policy checking system with the template based installation automation system, offering effective ways to help operators reduce the installation errors and increase the correctness assurance of application migration. Our experiments show that CloudMig can effectively detect a majority of the configuration errors in the migration process.

In the following sections, we discuss the potential causes that lead to the complicated and error-prone nature of the migration process in Section 2. We review the existing approaches and their limitations in Section 3. We report our operator-based case studies in Section 4 through a series of experiments conducted on migrating distributed system applications and multi-tier Internet services from local data center to Amazon EC2-like cloud, including the common migration problems observed and the key insights for solving the problems. In Section 5 we present the design of the CloudMig system, which provides both the configuration validation and installation automation to simplify the migration process.

## 2 Why Migration to Cloud is Complicated and Error-prone

There are some causes that make the migration process to Cloud complicated and error-prone. *First*, the computing environmental changes render many environment dependent configurations invalid. For example, as the database server is migrated from local data center to the Cloud, the IP address is possibly changed and this inevitably imposes the requirement of updating the IP address in all the components that depend on this database server. The migration process incurs large number of configuration update operations and even a single negligence of a single update may render the whole system out of operation. *Second*, the deployment of today's enterprise system consists of large number of different components. For example, for load balancing purpose, there may be multiple web servers and application servers in the systems. Thus the dependencies among the many components are rather complicated and can be broken very easily in the migration process. Sorting the dependency out to restore the normal operational status of the applications may take much more time than the migration process itself. *Third*, there are massive hidden controlling settings which may be broken inadvertently in the migration process. For example, the access controls of different components may be rumpled, which confront the system to the security threats. *Lastly*, the human operators in the complicated migration process may make many careless errors which are very difficult to identify. Overall, the complicated deployments, the massive dependencies, and the lack of automation make the migration process difficult and error-prone.

## 3 Related Work

Most of the existing migration approaches are either done manually or limited to only certain types of applications. For example, the suggestions recommended by Opencrowd are rather high level and abstract and lack the concrete assistances to the migration problem [4]. The solution provided by OfficetoCloud is only limited to the type of Microsoft Office products and does not even scratch the surface of large application migration [5]. We argue that a systematic and realistic study on the complexity of migrating large scale applications to Cloud is essential to direct the Cloud migration efforts. Furthermore, an automatic and operational approach is highly demanded to simplify and facilitate the migration process.

Nagaraja et al. [8] proposed a testbed for inserting faults to study the error behaviors. In our study, we study operator errors by migrating real practical applications from local data center to EC2. This forms a solid problem analysis context which motivates the effective solution for the migration problem. Vieira and Madeira [9] proposed to assess recoverability of database management systems through fault emulation and recovery procedure emulation. However, they assumed that human operators had the fault identification capability. In our work, we assume that human operators only have certain error identification capability but still cannot avoid errors.

Thus an automated configuration management system is highly demanded. There is already intensive research work on design and evaluation of interactive systems with human operators involved in the field of human computer interaction. For example, Maxion and Reeder in [7] studied the genesis of human operator errors and how to reduce them through user interface.

## 4 Migration Operations and Error Model

In this section, we describe a series of application migration practices conducted in migrating typical applications from a local data center to EC2 Cloud platform. We first introduce the experimental setup and then discuss the migration practices on representative applications in details and in particular, we focus on the most common errors made during the migration process. Based on our observations, we build the migration error model through the categorization of the migration errors.

### 4.1 Experiment Setup

Our experimental testbed involves both a local data center and EC2 Cloud. The local data center in College of Computing, Georgia Institute of Technology, is called "loki", which is a 12-node, 24-core Dell PowerEdge 1850 cluster. Because the majority of today's enterprise infrastructures are not virtualized, the physical to virtual (P2V) migration paradigm is the mainstream for migrating applications to virtualized cloud datacenters. In this work, we focus on P2V migration.

We deliberately selected representative applications as migration subjects. These applications are first deployed in the local data center and then operators are instructed to migrate from local data center to the Cloud. Our hypothesis is that application migration to the cloud is a complicated and error-prone process and thus a semi-automated migration validation system can significantly improve the efficiency and effectiveness of application migration. With the experimental setup across the local data center and Amazon EC2 platform, we are able to deploy moderate enterprise scale of applications for migration from a real local data center to the real Cloud platform and test the hypothesis under the setting of real workload, real massive systems, and real powerful Cloud.

We selected two types of applications in the migration case studies: Hadoop and RUBiS. These represent typical types of applications used in many enterprise computing systems today. The selection was made mainly by taking into account the service type, the architecture design and the migration content.

- Hadoop [1], as a powerful distributed computing paradigm, has been increasingly attractive to many enterprises to analyze large scale data generated daily, such as Facebook, Yahoo, etc. Many enterprises utilize Hadoop as a key component to achieve data intelligence. Because of its distributed nature, the more nodes participating in the computation, the more computation power is obtained in running Hadoop. Thus, when the computation resources are limited at local site, enterprises tend to migrate their data intelligence applications to Cloud to scale out the computation. From the aspect of service functionality, Hadoop is a very typical representation of data-intensive computation applications and thus the migration study on Hadoop provides us good referential value on data intensive application migration behaviors.

  Hadoop consists of two subsystems, map-reduce computation subsystem and Hadoop Distributed File System (HDFS), and thus migrating Hadoop from local data center to the Cloud includes both computation migration and file system migration or data migration. Thus it is a good example of composite migration. From the angle of architecture design, Hadoop adopts the typical master-slave structure in its two layers of subsystems. Namely, in map-reduce layer, a job tracker manages multiple task trackers and in the HDFS layer, a NameNode manages multiple DataNodes. Thus the dependency relationships among multiple system components form a typical tree structure. The migration study on Hadoop reveals the major difficulties or pitfalls in migrating applications with tree-style dependency relationships.

  In our P2V experiment setup, we deploy a 4-node physical Hadoop cluster, and designate one physical node to work as NameNode in HDFS or job tracker in map-reduce and four physical nodes as DataNode in HDFS or task tracker in map-reduce (the NameNode or job tracker also hosts a DataNode or task tracker). The Hadoop version we are using is Hadoop-0.20.2. The migration job is to migrate source Hadoop cluster to the EC2 platform into a virtual cluster with 4 virtual nodes.

- RUBiS [2] is an emulation of multi-tiered Internet services. We selected RUBiS as a representative case of large scale enterprise services. To achieve the scalability, enterprises often adopt the muti-tiered service architecture. Multiple servers are used for receiving Web requests, managing business logic, and storing and managing data: Web tier, application tier, and database tier. Depending on the workload, one can add or reduce the computation capability at a certain tier by adding more servers or removing some existing servers. Concretely, a typical three tier setup consists of using an Apache HTTP server, Tomcat application server and MYSQL database as the Web tier, application tier and database tier respectively.

  We selected RUBiS benchmark in our second migration case study by considering the following factors. First, Internet service is a very basic and prevalent application type in daily life. E-commerce enterprises such as EBay, usually adopts multi-tiered architecture as emulated by RUBiS to deploy their services and this renders RUBiS a representative case of Internet service architecture migration. Second, the dependency relationship among the tiers of multi-tiered services follows an acyclic graph structure, rather than a rigid tree structure, making it a good alternative in studying the dependency relationship preservation during the migration process. Third, the migration content of this type of application involves reallocation of application, logic and data and thus its migration provides a good case study on rich content migration. In the P2V experiment setup, one machine installs the Apache HTTPD server as the first

tier, and two machines install the Tomcat application server as the second tier, and two machines install the MYSQL database as the third tier.

In the following subsections, we introduce two migration case studies we have conducted: Hadoop migration and RUBiS migration, focusing mainly on configuration errors and installations errors. The configuration errors are our primary focus because they are the most frequent operator errors, some of which are also difficult to identify and correct. Installation errors can be corrected or eliminated by more organized installation steps or semi-automated installation tools with more detailed installation scripts and instructions.

We first discuss the typical configuration errors in each migration case study based on our error categorization model, which classifies the configuration errors into seven categories: dependency preservation error, network connectivity error, platform difference error, reliability error, shutdown and restart error, software and hardware compatibility error, and access control and security error. Then we describe the common installation errors across both case studies. We illustrate each category of errors by examples through selecting a subset of the typical errors observed in our experiments. Finally we present the statistical results on the error distributions in each case study and across case studies. This experimental analytic study of major errors lays a solid foundation for the design of a semi-automated migration validation system that offers effective configuration management.

### 4.2 Hadoop Migration Study

In the Hadoop migration case study, we migrate the source Hadoop application from the local data center to EC2 platform. This section discusses the typical configuration errors observed in this process.

**Dependency Preservation.** This is the most common error present in our experiments. Such a pitfall is very easy to make and very difficult to discover and may lead to disastrous results. According to the degree of severe impacts of this type of error on the deployment and migration, it can be further classified into four levels of errors.

The first level of errors is the "dependency preservation" error generated when the migration administrator fails to meet the necessity of dependency preservation checking. Even if the dependency information presents explicitly, lacking of enforcement to review the component dependency may lead to stale dependency information. For example, in our experiments, if the migration operator forgets to update the dependency information among the nodes in the Hadoop application, then the DataNodes (or task tracker) after migration will still initiate the connection with the old NameNode (or job tracker). This directly renders the system unoperational.

The second level of errors in Hadoop migration is due to incorrect formatting and typos in the dependency files. For example, a typo hidden in the host name or IP address renders some DataNodes to be unable to locate the NameNodes.

The third level of the dependency preservation error type is due to incomplete updates of dependency constraints. For example, one operator only updated the configuration files named "masters" and "slaves" which record the NameNode and list of DataNodes respectively. However, Hadoop dependency information is also located in some other configuration files such as "fs.default.name" in "core-site.xml" and "mapred.job.tracker" in mapred-site.xml. Thus Hadoop was still not able to boot with the new NameNode. This is a typical pitfall in migration, and is also difficult to detect by the operator because the operator may think that the whole dependency is updated and may spend intense efforts in locating faults in other locales.

The fourth level of the dependency preservation error type is due to inconsistency in updating the number of machines in the system. Often, an insufficient number of updated machines may lead to unexpected errors that are hard to debug by operators. For example, although the operator realizes the necessity to update the dependency constraints and also identifies all the locations of constraints on a single node, the operator may fail to update all the machines in the system, which are involved in the system-wide dependency constraints. For example, in Hadoop migration, if not all the DataNodes update their dependency constraints, the system cannot run with the participation of all the nodes.

**Network Connectivity** Bearing the distributed computing nature, Hadoop involves intensive communication across nodes in the sense that the NameNode keeps communication with DataNodes and job tracker communicates with task tracker continuously. Thus for such system to work correctly, inter-connectivities among nodes become an indispensible prerequisite condition. In our experiments, operators showed two types of network connectivity configuration errors after migrating Hadoop from the local data center to EC2 in the P2V migraton paradigm.

The first type of such error is that some operators did not set the network to enable all the machines to be able to reach each other over the network. For example, some operators forgot to update the file "/etc/hosts" and led to IP resolvement problems. The second type of such error is local DNS resolution error. For example, some operators did not set the local DNS resolution correctly, which led to the consequence that only the DataNodes residing in the same host as the master node were booted after the migration.

**Platform Difference** The platform difference between EC2 Cloud and local data center also creates some errors in migrating applications. These errors can be classified into three levels: *security*, *communication*, and *incorrect instance operation*. In our experiment, when the applications are hosted in the local data center, the machines are protected by the firewalls, and thus even if the operators set simple passwords, the security is complemented by the firewalls. However, when the applications are migrated into the public Cloud, the machine can experience all kinds of attacks and thus too simple passwords may render the virtual hosts susceptible to security threats. The second level of the platform difference error type is related to the communication setting difference between cloud and local data center. For example, such error may occur after the applications are migrated into EC2 Cloud, if the communication between two virtual instances is still set in the same way as if the applications were hosted in the local data center. Concretely, for the operator in one virtual instance to ssh another virtual instance, the identify file which is granted by Amazon must be provided. Without the identify file, the communication within virtual instance cannot be set correctly. The third level of the platform difference error type is rooted in the difference between virtual instance management infrastructures. In the experiments, there were operators who terminated an instance but his actual intention is to stop the instance. In EC2 platform, termination of an instance will lead to the elimination of the virtual instance from Cloud and thus all the applications installed and all the data stored within the virtual instance are lost if data is not backed up in persistent storage like Amazon Elastic Block storage. Thus, this poses critical risks on the instance operations, because a wrong instance operation may wipe out all the applications and data.

**Reliability Error**: In order to achieve fault tolerance and performance improvements, many enterprise applications like Hadoop and multi-tiered Internet services replicate its data or components. For example, in Hadoop, data is replicated in certain number of DataNodes, while in multi-tiered Internet services, there may exist multiple application servers or database servers. Thus after the migation, if the replication degree is not set correctly, either the migrated application fails to work correctly or the fault tolerance level is compromised. For example, in the experiments, there were cases in which the operator made errors that set the replication degree more than the total number of DataNodes in the system. The reliability errors are sometimes latent errors.

**Shutdown and Restart**: This type of error means that the shutdown or restart operation in the migration process may cause errors if not operating correctly. For example, a common data consistency error may occur if Hadoop is incorrectly shuts down the HDFS. More seriously,a shutdown or restart error sometimes may compromise the source system. In our experiment, when the dependency graph was not updated consistently and the source cluster was not shut down completely, the destination Hadoop cluster initiated to connect to the source cluster and acted as the client to connect to the source cluster. As a result, all the operations issued by the destination cluster actually manipulated the data in the source cluster and thus the source cluster data was contaminated. Such errors may create disastrous impacts on the source cluster and are dangerous if the configuration errors are not detected in time.

**Software and Hardware Compatibility**: This type of error is less common in Hadoop migration than in RUBiS migration partly because Hadoop is built on top of Java and thus has better interoperability and also Hadoop involves a relatively smaller number of different components than RUBiS. Sometimes, the difference in software versions may lead to errors. For instance, the initial Hadoop version selected by one operator was Hadoop 0.19, which showed bugs in the physical machine. After the operator turned to the latest 0.20.2 version, the issue disappeared.

**Access Control and Security**: It is noted that a single node Hadoop cluster can be set and migrated without root access. However, because a multi-node Hadoop cluster needs to change the network inter-connectivity and solve the local DNS resolution issue, the root access privilege is necessary. One operator assumed that the root privilege was not necessary for multi-node Hadoop installation and was blocked due to the network connectivity problem for about one hour and then sought help for access to the root privilege.

### 4.3 RUBiS Migration Study

In the RUBiS migration experiments, we migrate a RUBiS system with one web server and two application servers and two database servers from the local data center to EC2 Cloud. We below discuss the configuration errors present in the experiments in terms of the seven types of error categories.

**Dependency Preservation**: Similar to Hadoop migration, the dependency preservation error type is also the most common error in RUBiS migration. Because RUBiS has more intensive dependency among different components than Hadoop, operators made more configuration errors in the migration. For different tiers of a RUBiS system to run cooperatively, dependency constraints need to be specified explicitly in relevant configuration locales. For example, for each Tomcat server, its relevant information needs to be recorded in the configuration file named " workers.properties" in Apache HTTPD server. The MYSQL database server needs to be recorded in the RUBiS configuration file named "mysql.properties". Thus an error

in any of these dependency configuration files will lead to the operation error. In our experiments, operators made different kinds of dependency errors. For example, some operator migrated the application but forgot to update the Tomcat server name in workers.properties. As a consequence, although the Apache HTTPD server was running correctly, RUBiS was not operating correctly because the Tomcat server could not be connected. One operator could not find the configuration file location to update the MYSQL database server information in RUBiS residing in the same host as Tomcat and this led to errors and the operator therefore gave up the installation.

**Network Connectivity**: Relative to Hadoop migration, there is less node interoperability in a multi-tiered system like RUBiS, and different tiers present less needs on network connectivity, thus the network connectivity configuration errors are less frequently seen in RUBiS migration. One typical error was seen when the operator was connecting the Cloud virtual instance, he forgot to provide the identity file to enable two virtual instances to connect via ssh.

**Platform Difference** : This error type turns out to be a serious fundamental concern in RUBiS migration. Because sometimes the instance rebooting operation may change the domain name, public IP and internal IP, even if the multi-tiered service is migrated successfully, a rebooting operation may render the application to service interruption. One operator finished the migration and after fixed a few configuration errors, the application was working correctly in EC2. After we turned off the system on EC2 for one day and then rebooted the service, we found that because the domain name had totally changed, all of the IP addresses or host name information in configuration files needed to be updated.

**Reliability Error**: Due to the widely used replication in enterprise systems, it is typical that the system may have more than one application server and/or more than one database server. One operator spelt the name wrong for the second Tomcat server, but because there remained a working Tomcat server due to replication, the service was still going on without interruption. However, a hidden error as such was hidden inside the system and it may cause unexpected errors that could lead to detrimental damage and yet is hard to debug and correct. This further validates our argument that configuration error detection and correction tools are critical for cloud migration validation.

**Shutdown and Restart**: This type of error shows that incorrect server start or shutdown operation in multi-tiered services may render the whole service unavailable. For example, the Ubuntu virtual instance selected for the MYSQL tier has a different version of MYSQL database installed by default. One operator forgot to shut down and remove the default installation first before installing the new version of MYSQL and thus caused errors. The operator spent about half an hour to find the issuses and fixed them. Also we observed a couple of incidents where the operator forgot to boot the Tomcat server first before the shutdown operation, thus causing errors that are time consuming to debug.

**Software and Hardware Compatability**: this type of error also happens frequently in RUBiS migration. The physical machine is 64 bits, while one operator selected the 32 bits version of mod_jk ( the component used to forward the HTTP request from Apache HTTPD server to Tomcat server ) and thus incompatibility issues occured. The operator was stuck for about two hours, and finally identified the version error. After the software version was changed into 64 bits, the operator successfully fixed the error. A similar error was observed where an operator selected an arbitrary MYSQL version which took about one hour for the failed installation and then switched to a newer version before finally successfully installed the MYSQL database server.

**Access Control and Security**: This type of error also occurs frequently in RUBiS migration. For example, the virtual instance in EC2 Cloud bears the default feature of all ports closed. To enable the SSH operation possible, the security group where the virtual instance resides must open the corresponding port 22. Also one operator configured the Apache HTTPD server successfully but the Web server was unable to connect through port 80 and it took about 30 mins to identify the restrictions from EC2 documentation. Similar errors also happened for port 8080 which was for accessing Tomcat server. Another interesting error is that one operator set up the Apache HTTPD server, but forgot to set the root directory to be accessible and thus the index.html was not accessible. The operator reinstalled the HTTPD server but still did not discover the error. With the help of our configuration assistant, this operator finally identified the error and changed the access permission and fixed the error. We also found that operators also made errors in granting privileges to different users and one case was solved by seeking help in the MYSQL documentation.

### 4.4 Installation Errors

In our experiments-based case studies, we observe that operators may make all kinds of errors in installation or redeployment of the applications in Cloud. More importantly, these errors seem to be common across all types of applications. In this section we classify these errors into the following categories: **Context information error**: This is a very common installation error type. A typical example is that operators forget the context information they have used in the past installation. For example, the operators remembered the wrong path to install their applications and have to reinstall the applications from

scratch. Also if there are no automatic installation scripts or an incorrect or incomplete installation script is used, it can be a very frustrating experience with the same procedures repeated again and again. If the scale of the computing system is large, then the repeated installation process turns out to be a heavy burden for system operators. Thus a template based installation approach is highly recommended.

**Environment compatibility error** : In this migration case study, before any application can be installed, the computing environment compatibility needs to be ensured at both the hardware and software level. For example, there were migration failures created due to the small available disk space in virtual instance in migrating RUBiS. A similar errors is that the operator created a virtual instance with 32 bits operating system, while the application was a 64 bits version. Thus, it is necessary to check the environment compatibility before the application installation starts. An automatic environment checking process helps to reduce the errors caused by incorrect environment settings.

**Prerequisite resource checking error** : This type of error is originated from the fact that every application depends on a certain set of prerequisite facilities. For example, the installations of Hadoop and Tomcat server presume the installation of Java. In the experiments, we observed that the migration or installation process were prone to be interrupted by the ignorance of installing prerequisite standard facilities. For example, the compilation process needs to restart again due the lack of "gcc" installation in the system. Thus, a complete check-list of the prerequisite resources or facilities can help us reduce the interruptions of the migration process.

**Application installation error**: this error is the most common error type experienced by the operators. The concrete application installation process usually consists of multiple procedures. We found that the operator made many repeated errors even when the installation process for the same application was almost the same. For example, operators forgot the building location of the applications. Thus a template based application installation process will help facilitate the installation process.

### 4.5 Migration Error Distribution Analysis

In this section, we analyze the error distributions for each specific application and the error distribution across the applications.

Figure 1 and Figure 2 show the number of errors and percentage of error distribution in the Hadoop migration case study. In both figures, the X-axis indicates the error types as we analyzed in the previous sections. The Y-axis in Figure 1 shows the number of errors for each particular error type. The Y-axis in Figure 2 shows the share of each error type in terms of the percentage over the total number of errors. In this set of experiments, there were a total of 24 errors and some errors cause violation in multiple error categories. In comparison, the dependency preservation error happened most frequently. 42% of the errors belong to this error type with 10 occurrences. Operators typically made all four levels of dependency preservation errors as we discussed in Section 4.2. These kinds of errors took a long time for operators to detect. For example, an incomplete dependency constraint checking error took one operator two and a half hours to identify the cause of the error and fix it. Network connectivity error

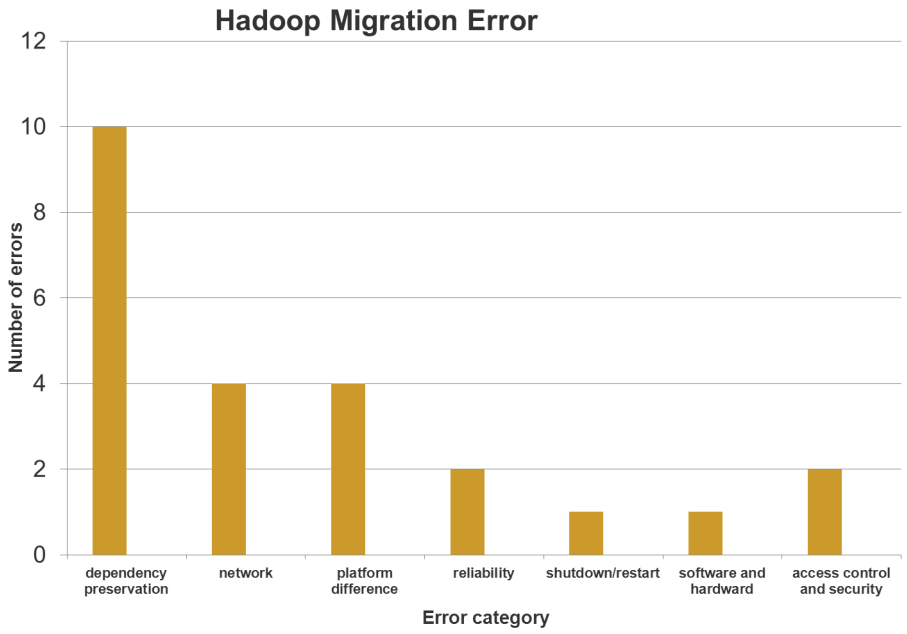


**Figure 1. Hadoop migration error**

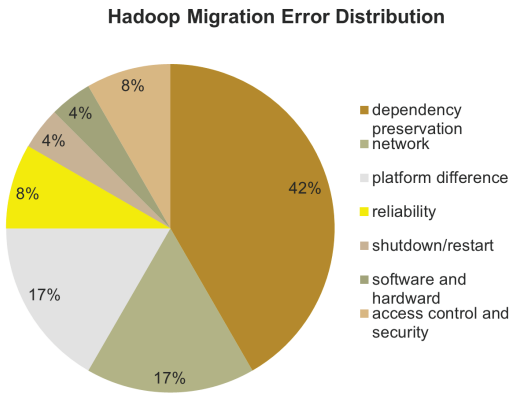


**Figure 2. Hadoop migration error distribution. The legend lists the error types in the decreasing frequency order.**

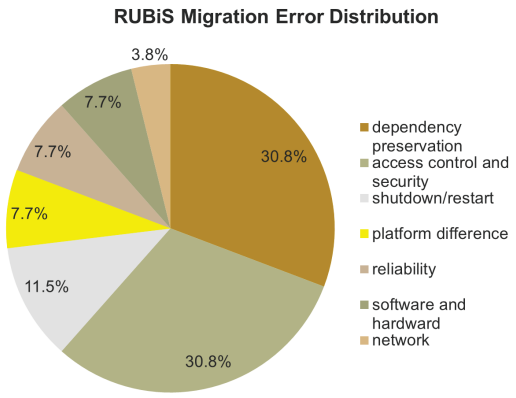


**Figure 3. RUBiS error distribution. The legend lists the error types in the decreasing frequency order.**

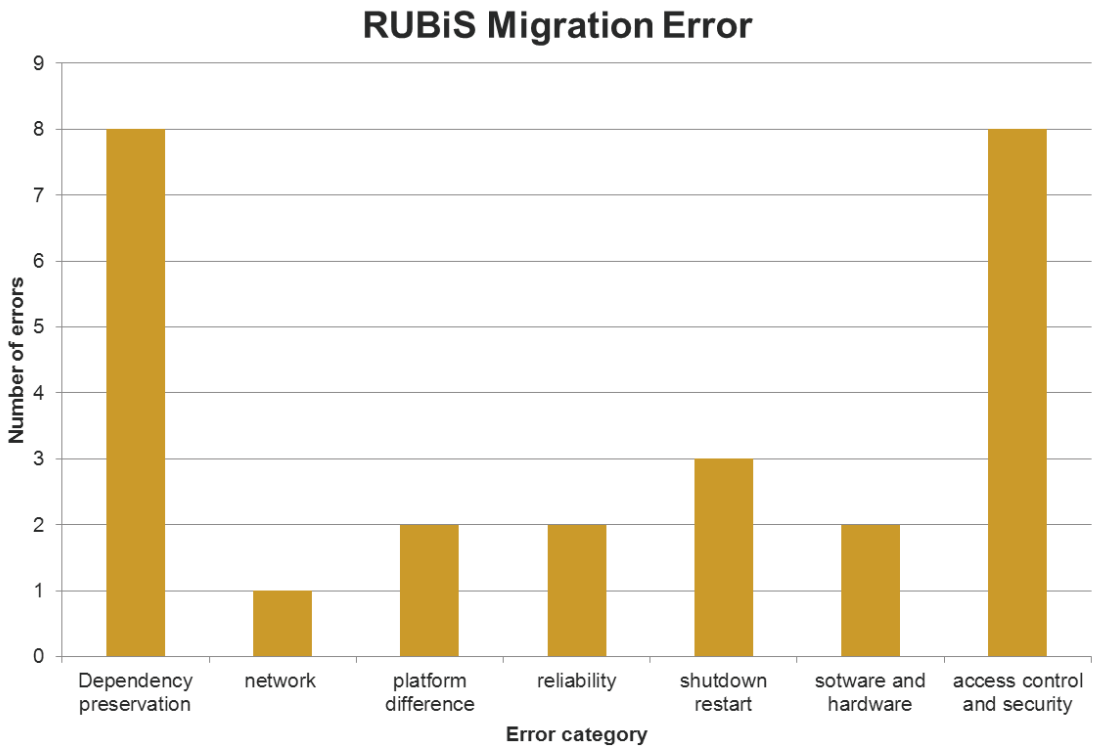


**Figure 4. RUBiS migration error**

and platform difference error were the next most frequent error types, each taking 17% of the total errors. Network connectivity errors included local DNS resolution and IP address update errors. One typical platform difference error was that the termination of an instance led to the data loss. Interesting to note is that these three types of errors take 76% of the total errors and are the dominating types of the error occurrences observed in the experiments we conducted.

Figure 4 and Figure 3 show the number of error occurrences and the percentage of error distribution for RUBiS migration case study respectively. There were a total of 26 error occurrences observed in this process and some errors fall into several

error categories. The dependency preservation error and access control and security errors were the two most frequent error types, each with 8 occurrences, taking 31% of the total erorrs. Together, both error types covered 62% of all the errors and dominated the error occurrences. It is interesting to note that the distribution of errors in the RUBiS migration case study was very different from the distribution in the Hadoop migration case study. For example, the number of access and security errors in RUBiS was 4 times the number of errors of this type in Hadoop migration. This is because RUBiS migration demanded the correct access control settings for many more entities than Hadoop. Not surprisingly, the majority of the access control errors were file access permission errors. This is because changing the file access permission is a common operation in setting up web services and sometimes operators forgot to validate whether the access permissions were set correctly or not. Also when there were errors and the system could not run correctly, the operators often ignored the possibility of this type of simple errors and thus led to longer time spent on error identification. For example, one error of this type took more than 1 hour to identify. Also there were more ports to open in RUBiS migration than in Hadoop migration, which also led to the high frequency of access control errors in RUBiS migration. RUBiS migration presented more software and hardware compatibility errors than Hadoop migration because the number of different components that were involved in RUBiS application is, relatively speaking, much more than in the typical Hadoop migration. Similarly, there were more "shutdown/restart" errors in the RUBiS migration. On the other hand, Hadoop migration presented more network connectivity errors and platform difference errors than RUBiS migration, because Hadoop nodes require more tightly coupled connectivity than the nodes in RUBiS. For example, the master node needs to have direct access without password control to all of its slave nodes.

Figure 5 and Figure 6 summarize across Hadoop migration and RUBiS migration case studies by showing the number of error occurrences and the percentage of error distribution, respectively.The dependency preservation errors are the most frequent error occurrences and accounted for 36% of the total errors. In practice, this was also the type of error that on average took the longest time to identify and fix. This is primarily because dependency constraints are widely distributed among system configurations, it is very prone to be broken by changes to the common configuration parameters. The second biggest error source was the "access control and security" errors, which accounted for 20% of the total number of error occurences. It was very easy for operators to change the file permissions to incorrect settings or some other habits which were fitting in local data center might render the application susceptible to security threats in the Cloud environment. The operational or environmental differences between Cloud and local data centers formed the third largest source of error, accounting for 12% of all the errors. Many common operations in local data center might lead to errors in Cloud if no adjustments to Cloud environment were made. These three types of errors dominated the error distribution, and accumulatively accounted for 68% of the total errors. In addition to these three types of errors, network connectivity was also an important source of errors, accounting for 10% of the total errors, because of the heavy inter-nodes operations in many enterprise applications today. The rest of errors accounted for 32% of the total errors. These error distributions provide a good reference model for us to build a solid testbed to test the design of our CloudMig migration validation approach to be presented in the subsequent sections of this paper. We argue that a cloud migration validation system should be equipped with an effective configuration management component that not only provides a mechanism to reduce the configuration errors, but also equips the system with active configuration error detection and debugging as well as semi-automated error correction

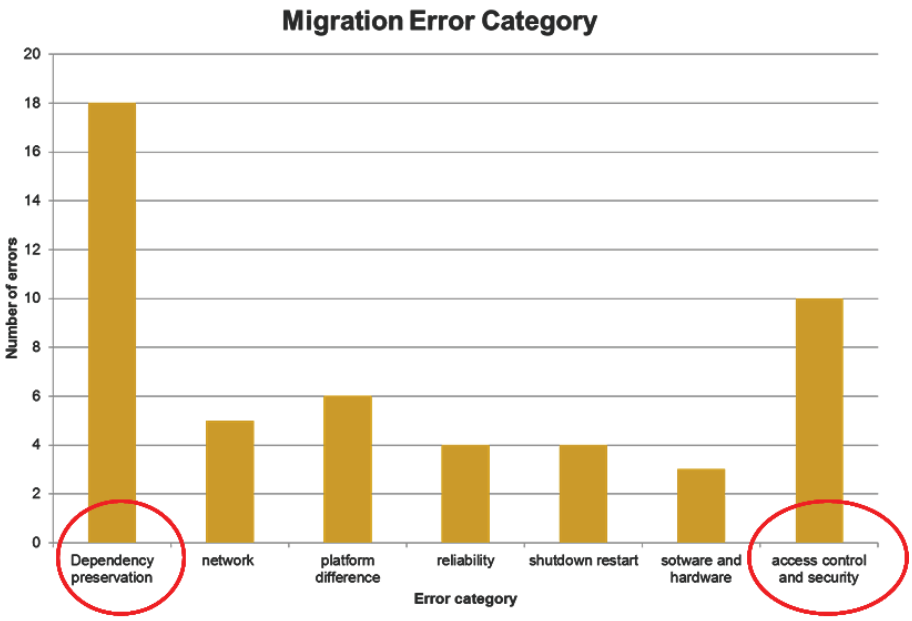


**Figure 5. Overall migration error**

and repairs.

## 5 Migration Validation with CloudMig

The case studies showed that the installation mistakes and configuration errors were the two major sources of errors in migrating applications from local data centers to Cloud. Thus a migration management framework is highly recommended to provide the installation automation and configuration validation. We present the design of CloudMig, a semi-automated configuration management system, which utilizes a "template" to simplify the large scale enterprise system installation process and utilizes a "policy" as an effective means to capture configuration dependency constraints, validate the configuration correctness, and monitor and respond to the configuration changes.

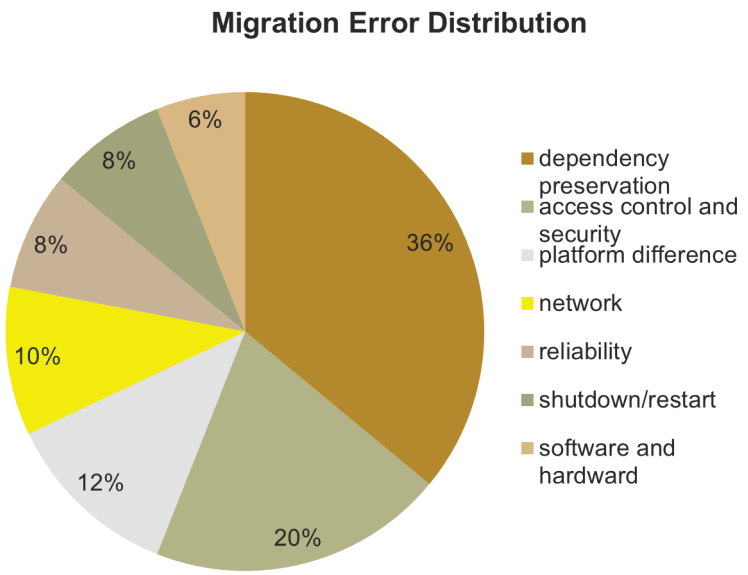


**Figure 6. Overall migration error distribution. The legend lists the error types in the decreasing frequency order.**

The architecture design of the CloudMig system aims at coordinating the migration process across different data centers by utilizing template-based installation procedures to simplify the migration process and utilizing policy-based configuration management to capture and enforce configuration related dependency constraints and improve migration assurance.

The first prototype of CloudMig configuration management and validation system consists of four main components: the centralized configuration management engine, the client-based local configuration management engine, the configuration template management tool and the configuration policy management tool. The template model and the configuration policy model form the core of CloudMig for semi-automated installation and configuration validation system. In the subsequent sections we will briefly describe the functionality of each of these four components.

### 5.1 Configuration Template Model

CloudMig uses a template as an effective mechanism to simplify the installation process. Template is a pre-formatted script-based example file containing place holders for dynamic and application-specific information to be substituted at application migration time for concrete use.

In CloudMig, the installation and configuration management is operating in the unit of the application. That is, each application corresponds to a template set and a validation policy set. The central management server is responsible to manage the collection of templates and configurations on a per application basis and provides migration planning for the migration process.

Recall that in the observations obtained from our migration experiments in Section 4, one big obstacle and source of errors in application migration is the installation and configuration process which is also a recurring process in system deployment and application migration. We propose to use the template approach to reduce the complexities of the installation process and reduce the chances of errors. An installation template is defined by an installation script with place holders for dynamic and application specific information. Templates simplify the recurring installation practice of particular applications by

substituting the dynamic information with new values. For example, in an enterprise system with 100 nodes, there will be multiple applications ranging from MYSQL database nodes, Tomcat application server nodes, to Hadoop distributed system nodes and so forth. Distributed applications may span and extend to more nodes on demand to scale out. For each application, its installation templates are stored in the installation template repository. These templates are sorted by the application type and an application identifier. The intuitive idea of template is that through information abstraction, the template can be used and refined for many similar nodes through parameter substitution to simplify the installation process for large scale systems. For example, if a Hadoop system consists of 100 DataNodes, then only a single installation template is stored in the installation template repository and each DataNode will receive the same copy of the installation template with only parameter substitution efforts needed before running the installation scripts to set up the DataNode component in each individual node. The configuration dependency constraints are defined in the policy repository to be described in the next subsection. CloudMig classifies the templates into the following four types:

1. Context dictionary: This is the template specifying the context information about the application installation. For example, the installation path, the preassumed Java package version, etc. A context dictionary template can be as simple as a collection of the key-value pairs. Users specify the concrete values before a particular application installation. Dynamic place holders for certain key context information achieve the installation flexibility and increase the ability to find out the relevant installation information in the presence of system failures.

2. Standard facility checklist template: This is the script template to check the prerequisites to install the application. Usually these are some standard facilities, such as Java or OpenSSH. Typical checklists include those for verifying the Java path setting, checking installation package existence, and so on. These checklists are common to many applications and are prerequisites for the success of installing the applications and thus performing a template check before the actual installation can effectively reduce the errors caused by ignorance of the checklist items. For example, both Hadoop and Tomcat server rely on the correct Java path setting and thus the correct setting of Java path is the prerequisite of successfully installing these two applications. In CloudMig, we collect and maintain such templates in a template library, which is shared by multiple applications. Running the checklist validation check can effectively speed up the installation process by reducing the amount of errors caused by carelessness on prerequisites.

3. Local resource checklist template: This is the script template to check the hidden conditions for an application to be installed. A typical example is to perform the check of whether or not there is enough available disk space quota for a given application. Similarly, such resource checklist templates are also organized by application type and application identifier in the template library and utilized by the configuration management client to reduce the local installation errors and installation delay.

4. Application installation template: This is the script template used to install a particular application. The context dictionary is included as a component of the template. Organizing installation templates simplifies the installation process and thus reduces the overhead in recurring installations and migration deployments.

## 5.2 Configuration Policy Model

In this section, we first introduce the basic concept of configuration policy, which plays the key role in capturing and specifying configuration dependency constraints and monitoring and detecting configuration anomalies in large enterprise application migration. Then we introduce the concept of continual query (CQ) and the design of a CQ enabled configuration policy enforcement model.

### 5.2.1 Modeling Dependency Constraints with Configuration Policies

A configuration policy defines an application-specific configuration dependency constraint. Here is an example of such constraints for RUBiS: for each Tomcat server, its relevant information needs to be specified explicitly in the configuration file named “workers.properties” in Apache HTTPD server. Configuration files are usually application-specific and usually specify the settings of the system parameters, the dependencies among the system components and thus directly impact the way of how the system is running. As enterprise applications scale out, the number of components may increase rapidly and the correlations among the system components evolve with added complexity. In term of complexity, configuration files for a large system may cover many aspects of the system configuration, ranging from host system information, to network setting, to security protocol and so on. Any typo or error may disable the operational behavior of the whole application system

as we showed and analyzed in the previous experiments. Configuration setting and management are usually a long term practice, starting from the time when the application is set up until the time when the application is ceased its use. During this long application life cycle, different operators may be involved in the configuration management practices and operate on the configuration settings based on their understandings, thus it further increases the probability of errors in configuration management. In order to fully utilize resources, enterprises may bundle multiple applications to run on top of a single physical node, and the addition of new applications may necessitate the need to change the configurations of existing applications. Security threats such as viruses, also pose demands to effective configuration monitoring and management.

In CloudMig, we propose to use policy as an effective means of ensuring the constraints of configurations to be captured correctly and enforced consistently. A policy can be viewed as a specialized tool to specify the constraints on the configuration of a specific application. It specifies the constraints to which the application configuration must conform in order to assure that the whole application is migrated correctly to run in the new environment. For example, in the Hadoop system, the configuration constraint that "the replication degree cannot exceed the number of DataNodes" can be represented as a Hadoop specific configuration policy. The basic idea of introducing the policy-based configuration management model is that if operators are equipped with a migration configuration tool to define the constraints that configuration must follow in the form of policies, then running the policy enforcement checks at a certain frequency will help to detect and eliminate certain types of errors, even although errors are unavoidable. Here are a few configuration policy examples that operators may have in migrating a Hadoop system.

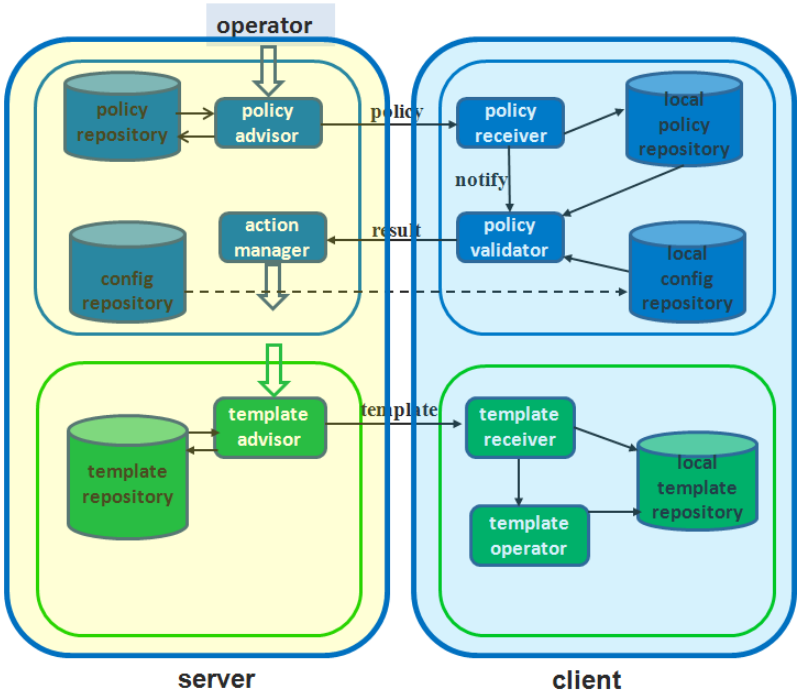


**Figure 7. CloudMig Architecture Design**

1. The replication degree can not be larger than the number of DataNodes
2. There is only one master node
3. The master node of Hadoop cluster should be named "dummy1"
4. The task tracker node should be named "dummy2"

As the system evolves and the configuration repository grows, performing such checking manually will become a heavy and error-prone process. For example, in enterprise Internet service systems, there may be hundreds of nodes, and the configuration of each node needs to follow certain constraints. For load balancing purpose, different Apache HTTPD servers correspond to different sets of Tomcat servers. Incorrect setting of relevant configuration entries will directly lead to an unbalanced system and even cause the system to crash when workload burst happens. With thousands of configuration entries, hundreds of nodes, and many applications, it is impractical if not impossible to perform manual configuration correctness checking and error correction. We argue that a semi-automated configuration constraint checking framework can greatly simplify the migration configuration and validation management of large scale enterprise systems. In CloudMig, we advocate the use of continual query as the basic mechanism for automating the configuration validation process of operator-defined configuration policies. In the next section we will describe how CQ-enabled configuration policy management engine can improve the error detection and debugging efficiency, thus reducing the complexity of migrating applications from a local data center to Cloud.

#### 5.2.2 Continual Query Based Policy Model

In CloudMig, we propose a continual query based policy specification and enforcement model. A continual query (CQ) [6] is defined as a triple in the form of (Query, Trigger, Stop). A continual query (CQ) can be seen as a standing query, in which the trigger component specifies the monitoring condition and is being evaluated periodically upon the installation of the CQ and whenever the trigger condition is true, the query component will be executed. The Stop component defines the condition to terminate the execution of the CQ. Trigger condition can be either time-based or content-based, such as "checking the free disk space every hour or trigger a configuration action when the free disk space is less than 1GB".

In CloudMig, we define a policy in the format of continual query and refer to the configuration policy as the Contiual Query Policy (CQP), denoted by : CQP(policyID, appName, query, trigger, action, stopCondition). Each element of the CQP is defined as follows:

1. *policyID* is the unique numeric identifier of the policy.
2. *appName* is the name of the application that is installed or migrated to the host machine.
3. *query* refers to the search of matching policies and the execution of policy checking. The query can be a Boolean expression over a simple key-value repository or SQL-like query or XQuery on a relational database of policies.
4. *trigger* is the condition upon which the policy query will be executed. Triggers can be classified into time-based or content-based.
5. *action* indicates the action to be taken upon the query results. It can be a warning flag in the configuration table or an warning message sent by email or displayed on the command line of an operator's terminal.
6. *stopCondition* is the condition upon which the CQP will stop to execute.

An example CQ-based policy is to check whether the replication degree is larger than the number of DataNodes in Hadoop prior to migration or changing the replica factor (replication degree) or reducing the number of DataNodes. Whenever the check returns a true value, send an alert to re-configure the system. Clearly, the query component is responsible for checking if the replication degree is larger than the number of DataNodes in Hadoop. The trigger condition is Hadoop migration or changing the replica factor (replication degree) or reducing the number of DataNodes. The action is defined as re-configuration of the Hadoop system upon the true value of the policy checking. In CloudMig, we introduce default stop condition of one month for all CQ-enabled configuration policies.

### 5.3 CloudMig Server side Template Management and Policy Management

CloudMig aims at managing the installation templates and configuration policies to simplify the migration for large scale enterprise systems which may be comprised of thousands of nodes with multi-tier applications. Each application has its own configuration policy set and installation template set and the whole system needs to manage a large collection of configuration policies and installation templates. The CloudMig server side configuration management system helps to manage the large collection of templates and configuration policies effectively by providing system administrators (operators) with convenient tools to operate on the templates and policies. Typical operations include policy or template search, indexing, application specific packaging and shipping, to name a few. Detaching the template and policy management from individual application and utilizing a centralized server also improves the reliability of CloudMig in the presence of individual node failures.

In CloudMig, the configuration management server operates at the unit of a single application. Each application corresponds to an installation template set and a configuration validation policy set. The central management server is responsible for managing the large collection of configurations on a per application basis and providing migration planning to speed up the migration process and increase the assurance of application migration. Concretely, the configuration management server mainly coordinate the tasks of the installation template management engine and the configuration policy management engine.

**Installation Template Management Engine.**

As shown in Figure 7, the installation template management engine is the system component which is responsible for creating template, update template, advise the template for installation. It consists of a central template repository and a template advisor. The template repository stores and maintains the template collections of all the applications. The template advisor provides the operators with the template manipulation capabilities such as creating, updating, deleting, searching and

**Table 1. Migration Error Detection Rate**

| Migration Type | Error Detection Rate |
|---|---|
| Hadoop migration | 83% |
| RUBiS migration | 55% |
| all migrations | 70% |

indexing templates over the template repository. On a per application basis, operators may create an application template set, add new templates to the set, update templates from this set or delete templates. The template advisor assumes the job to search and dispatch templates for new installations and propagate template updates to corresponding application hosting nodes. For example, during the process of RUBiS installation, for a specific node, the template advisor dispatches the appropriate template depending on the server type (web server, application server or database server) and transmits (ships) the new installation set to the particular node.

Concretely, for each application, the central installation template management engine builds the context library which stores all the source information in the key-value pairs, and selects a collection of standard facility checklist templates which apply to the particular application, and pick a set of local resource checklist templates as the checklist collection for the application, and finally builds the specific application installation template. The central management engine then bundles the collections of templates and policies for the particular application and transmits the bundle to the client installation template manager to start the installation instance.

**Configuration Policy Management Engine.**
As the central management unit for the policies, the policy engine consists of four components: policy repository, configuration repository, policy advisor, and action manager. Together they cooperate to provide the service to create, maintain, dispatch and monitor policies and execute the corresponding actions based on the policy execution results. Concretely, we below describe the different components of the policy engine:

1. The policy repository is the central store where all the policies for all the applications are maintained. It is also organized on a per application basis. Each application corresponds to a specific policy set. This policy set is open to addition, update, or delete operations. Each policy corresponds to a constraint set on the application.

2. The policy advisor works on the policies in the policy repository directly and provides the functionalities for application operators to express the constraints in the form of CQ-based policy. Application operators creates policies through this interface.

3. The configuration repository stores all the configuration files on a per application basis. It ships the configurations from the CloudMig configuration server to the local configuration repository on the individual node (cient) of the system.

4. The action manager handles the validation results from the policy validator running on client and triggers the corresponding action based on certain policy query result, in the form of an alert through message posting or email or other notification methods.

## 5.4 CloudMig Configuration Management Client

The CloudMig configuration management client is running at each node of a distributed or multi-tier system, which is responsible for managing the configuration policies related to the node locally. Corresponding to the CloudMig configuration management engine at the server side, CloudMig client works as a thin local manager for the templates and policies which only apply to a particular node. A client engine mainly consists of two components: client template manager and client policy manager.

**Client Template Manager.**
Client template manager manages the templates for all the applications installed in the host node on per application basis. It consists of three components: template receiver, template operator and local template repository. The template receiver receives the templates from the remote CloudMig configuration management server and delivers the templates to local template manager. The local template manager installs the application based on the template with necessary substitution operations. The local template manager is also responsible for managing the local template repository which stores all the templates for the applications that reside at this node.

The concrete process of template based installation works as follows: after the client template manager receives the collection of installation templates from the server side installation template management engine, it will run the local resource checklist templates first to detect if there are any prerequisite checklist items which are not met. For example, it checks if the available disk space is less than the amount needed to install the application, or if the user has the access permissions to the installation path, etc. Next, the standard facility checklist template will run to detect if all the standard facilities are installed or not. Finally, the dynamic information in application specific templates are substituted and the context dictionary is integrated to run this normal installation process.

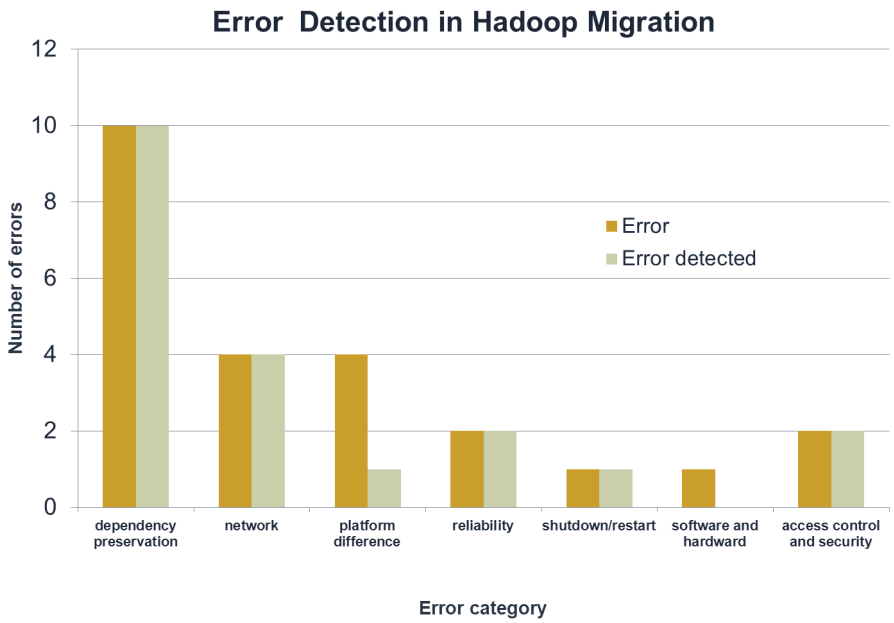


**Figure 8. Hadoop error detection**

**Client Policy Manager.**

There is a client policy manager residing together with the host node to manage the policies for the local node. It mainly consists of policy receiver, policy validator, local policy repository and local config repository. The policy receiver receives the policies transmitted from the policy advisor in the central policy server, and stores the policies in the local policy repository. The local config repository receives the configuration data directly from the central config repository. The local policy validator runs each policy. It retrieves the policy from local policy repository and searches the related configuration data to run the policy upon the configuration data. The policy validator transmits the validation results to the action manager in

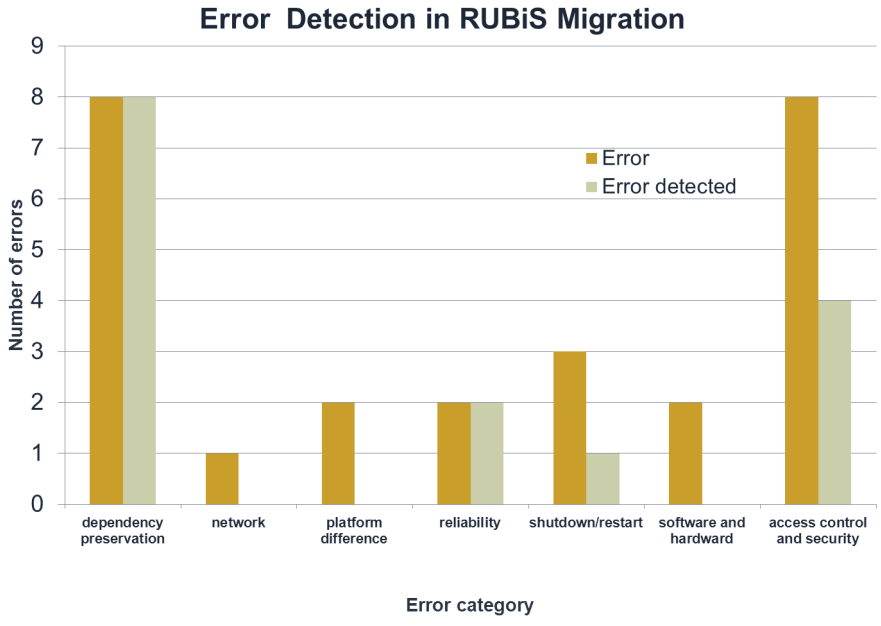


**Figure 9. RUBiS error detection**

the central server to take the alert actions.

## 6 Case Studies with CloudMig

We run CloudMig with the same set of operators on the same set of case studies after the manual migration process is done (recall Section 4). We count the number of errors that are detected by CloudMig configuration management and installation automation system. We show through a set of experimental results below that CloudMig overall achieves high error detection rate.

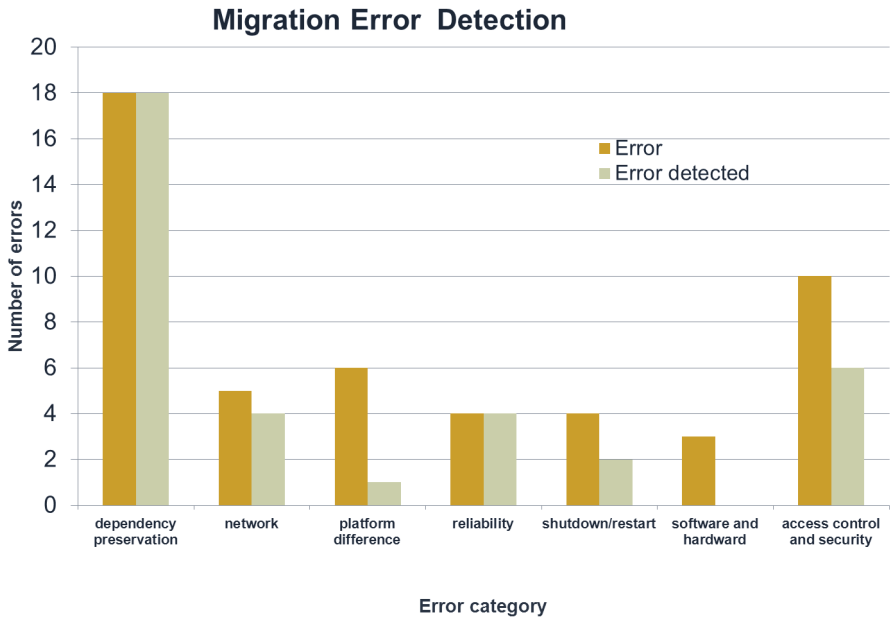


**Figure 10. Overall migration error detection**

Figure 8 shows the error detection results for Hadoop migration case study. As one can see, the configuration checking system can detect all the dependency preservation errors, network connectivity errors, shutdown restart errors, and all the access control errors. This confirms the effectiveness of the proposed system, in that it can detect the majority of the configuration errors. The two types of error that can not be fully detected are platform difference error and software/hardware compatibility errors. For platform difference errors, this is because the special property of the platform difference error requires the operators to fully understand the uniqueness of the particular Cloud platform first. As long as the operator understands the platform sufficiently, for example, by lessons learned from others or policies shared by others, we believe that such errors can be reduced significantly as well. The reason that current implementation of CloudMig cannot detect software/hardware compatibility errors notably is due to the quality of the default configuration data which lacks of application-specific software/hardware compatibility information. Although in the first phase of implementation, we mainly focus on the configuration checking triggered by the original configuration data, we believe that as operators weave more compatibility policies into CloudMig policy engine, such type of errors can also be reduced significantly. As Table 1 shows, totally CloudMig could detect $83\%$ of the errors in Hadoop migration.

Figure 9 shows the error detection result for RUBiS migration case study. In this study, we can see that CloudMig can detect all the dependency preservation errors and reliability errors.

However, because multi-tiered Internet service system involves a higher number of different applications, it leads to more complicated software/hardware compatibility issues compared to the case of Hadoop migration. In the experiments reported in this paper we are focusing on the configuration driven by the default configuration policies, which lacks of adequate software/hardware compatibility policies for RUBiS, thus CloudMig system did not detect the software/hardware errors. On the other hand, this result also indicates that in the RUBiS migration process, the operators are suggested to pay special attention to the software/hardware compatibility issues because such errors are difficult to detect with automated tools. It is interesting to note that the CloudMig was able to detect only half of the access control errors in RUBiS. This is because these errors include MYSQL privilege grant operations which are embedded in the application itself and the CloudMig configuration validation tool cannot intervene with the internal operations of MYSQL. Overall, CloudMig detected $55\%$ of the errors in RUBiS migration as shown in Table 1.

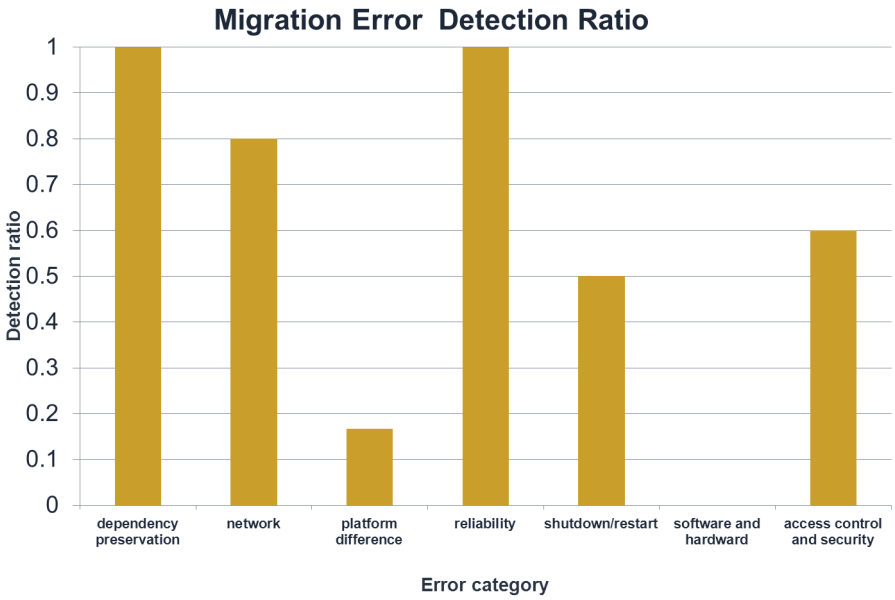


**Figure 11. Overall migration error detection ratio**

Figure 10 and Figure 11 show the number of detected errors and the error detection ratio of each error type summarized across the Hadoop migration case study and RUBiS migration case study respectively. Overall, CloudMig can detect all the dependency preservation and reliability errors and 80% of the network errors and 60% of the access control and security errors. In total, these four types of errors accounted for 74% of the total error occurrences. For shutdown/restart errors, CloudMig detected 50% of such errors and did not detect the software/hardware compatibility errors. This is because the application configuration data usually contains less information related with shutdown/restart operations or software/hardware compatibility constraints and this fact makes the configuration checking on these types of errors difficult without adding additional configuration policies. Figure 12 shows the percentage of error types in the total number of detected errors. One can see that 51% of the detected errors are dependency preservation errors, and 17% of the detected errors are network errors. Table 1 shows that totally across all the migrations, the error detection rate of CloudMig system is 70%.

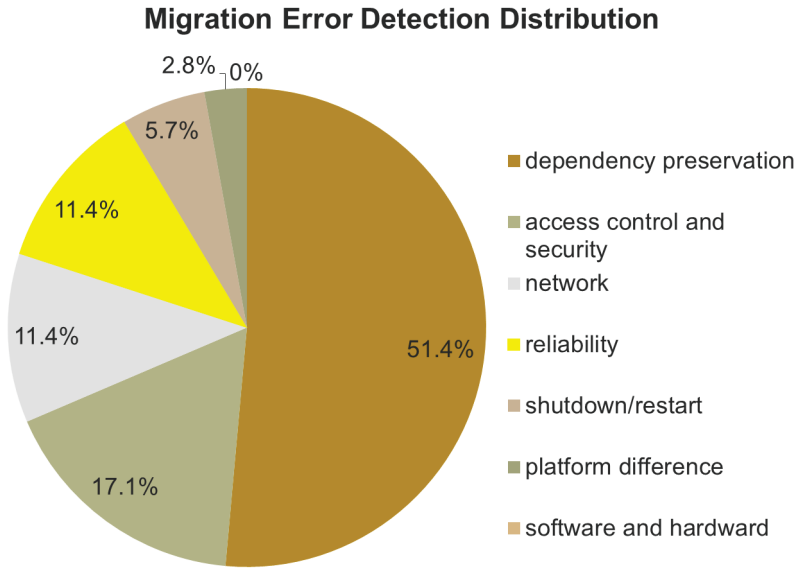


**Figure 12. Overall migration error detection percentage. The legend lists the error types in the decreasing percentage order.**

Overall these experimental results show the efficacy of CloudMig in reducing the migration configuration errors, simpli-

fying the migration process and increasing the level of assurance of migration correctness.

## 7 Conclusion

We have discussed the system migration challenge faced by enterprises in migrating local data center applications to the Cloud platform. We analyze why such migration is a complicated and error-prone process and pointed out the limitations of the existing approaches to address this problem. Then we introduce the operator-based experimental study conducted over two representative systems (Hadoop and RUBiS) to investigate the error sources. From these experiments, we build the error classification model and analyze the demands for an semi-automated configuration management and migration validation system. Based on the operator study, we design the CloudMig system with two unique characteristics. First, we develop a continual query based configuration policy checking system, which facilitate operators to weave important configuration constraints into continual query policies and periodically run these policies to monitor the configuration changes and detect and alert the possible configuration constraints violations. In addition, CloudMig combines the continual query based policy checking system with the template based installation automation system, offering effective ways to help operators reduce the installation errors and increase the correctness assurance of application migration. Our experiments show that CloudMig can effectively detect a majority of the configuration errors in the migration process.

## 8 Aknowlegement

This work is partly sponsored by grants from NSF CISE NetSE program, CyberTrust program, Cross-cutting program and an IBM faculty award, an IBM SUR grant and a grant from Intel Research Council.

## References

[1] Hadoop project. http://hadoop.apache.org/.
[2] RUBiS benchmark. http://rubis.ow2.org/.
[3] Amazon EC2. http://aws.amazon.com/ec2/, April 2011.
[4] Cloud Migration. http://www.opencrowd.com/services/migration.php, April 2011.
[5] Office Cloud. http://www.officetocloud.com, April 2011.
[6] L. Liu, C. Pu, and W. Tang. Continual queries for internet scale event-driven information delivery. *Knowledge and Data Engineering, IEEE Transactions on*, 11(4):610 –628, jul/aug 1999.
[7] R. A. Maxion and R. W. Reeder. Improving user-interface dependability through mitigation of human error. *Int. J. Hum.-Comput. Stud.*, 63:25–50, July 2005.
[8] K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen. Understanding and dealing with operator mistakes in internet services. In *In Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI 04*, 2004.
[9] M. Vieira and H. Madeira. Recovery and performance balance of a cots dbms in the presence of operator faults. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, DSN '02, pages 615–626, Washington, DC, USA, 2002. IEEE Computer Society.