Why Do Upgrades Fail And What Can We Do About It?

Toward Dependable, Online Upgrades in Enterprise System

Tudor Dumitraş and Priya Narasimhan

Carnegie Mellon University, Pittsburgh PA 15213, USA
tudor@cmu.edu, priya@cs.cmu.edu

Abstract. Enterprise-system upgrades are unreliable and often produce downtime or data-loss. Errors in the upgrade procedure, such as broken dependencies, constitute the leading cause of upgrade failures. We propose a novel upgrade-centric fault model, based on data from three independent sources, which focuses on the impact of procedural errors rather than software defects. We show that current approaches for upgrading enterprise systems, such as rolling upgrades, are vulnerable to these faults because the upgrade is not an atomic operation and it risks breaking hidden dependencies among the distributed system-components. We also present a mechanism for tolerating complex procedural errors during an upgrade. Our system, called Imago, improves availability in the fault-free case, by performing an online upgrade, and in the faulty case, by reducing the risk of failure due to breaking hidden dependencies. Imago performs an end-to-end upgrade atomically and dependably, by dedicating separate resources to the new version and by isolating the old version from the upgrade procedure. Through fault injection, we show that Imago is more reliable than online-upgrade approaches that rely on dependency-tracking and that create system states with mixed versions.

1 Introduction

Software upgrades are unavoidable in enterprise systems. For example, business reasons sometimes mandate switching vendors; responding to customer expectations and conforming with government regulations can require new functionality. Moreover, many enterprises can no longer afford to incur the high cost of downtime and must perform such upgrades online, without stopping their systems. While fault-tolerance mechanisms focus almost entirely on responding to, avoiding, or tolerating unexpected faults or security violations, system unavailability is usually the result of planned events, such as upgrades. A 2007 survey of 50 system administrators from multiple countries (82% of whom had more than five years of experience) concluded that, on average, 8.6% of upgrades fail, with some administrators reporting failure rates up to 50% [1]. The survey identified broken dependencies and altered system-behavior as the leading causes of upgrade failure, followed by bugs in the new version and incompatibility with legacy configurations. This suggests that most upgrade failures are not due to software defects, but to *faults that affect the upgrade procedure*.

For instance, in August 1996, an upgrade in the main data center of AOL—the world's largest Internet Service Provider at the time—was followed by a 19-hour outage. The system behavior did not improve even after the upgrade was rolled back, because the routing tables had been corrupted during the upgrade [2]. In November 2003, the upgrade of a customer relationship management (CRM) system at AT&T Wireless created a ripple effect that disabled several key systems, affecting 50,000 customers per week. The complexity of dependencies on 15 legacy back-end systems was unmanageable, the integration could not be tested in advance in a realistic environment, and rollback became impossible because enough of the old version had not been preserved. The negative effects lasted for 3 months with a loss of $100 M in revenue, which had dire consequences for the future of the company [3]. In 2006, in the emergency room of a major hospital, an automated

drug dispenser went offline, after the upgrade of a separate system, preventing a patient in critical condition from receiving the appropriate medication [4].

Existing upgrade techniques rely on tracking the complex dependencies among the distributed system components. When the old and new versions of the system-under-upgrade share dependencies (*e.g.*, they rely on the same third-party component but require different versions of its API), the upgrade procedure must avoid breaking these dependencies in order to prevent unavailability or data-loss. Because dependencies cannot always be inferred automatically, upgrade techniques rely on metadata that is partially maintained by teams of developers and quality-assurance engineers through a time-intensive and error-prone manual process. Moreover, the problem of resolving the dependencies of a component is NP-complete [5], which suggests that the size of dependency-repositories will determine the point at which ensuring the correctness of upgrades by tracking dependencies becomes computationally infeasible.

Because the benefits of dependency-tracking are reaching their limit, industry best-practices recommend "rolling upgrades," which upgrade-and-reboot one node at a time, in a wave rolling through the cluster. Rolling upgrades cannot perform incompatible upgrades (*e.g.*, changing a component's API). However, this approach is believed to reduce the risks of upgrading because failures are localized and might not affect the entire distributed system [6, 7].

In this paper, we challenge this conventional wisdom by showing that atomic, end-to-end upgrades provide more dependability and flexibility. Piecewise, gradual upgrades can cause global system failures by breaking *hidden dependencies*—dependencies that cannot be detected automatically or that are overlooked because of their complexity. Moreover, completely eliminating software defects would not guarantee the reliability of enterprise upgrades because faults in the upgrade procedure can lead to broken dependencies. We make three contributions:

- We establish a rigorous, upgrade-centric fault model, with four distinct categories: (1) simple configuration errors (*e.g.*, typos); (2) semantic configuration errors (*e.g.*, misunderstood effects of parameters); (3) broken environmental dependencies (*e.g.*, library or port conflicts); and (4) data-access errors, which render the persistent data partially-unavailable. §2
- We present Imago[1] (Fig. 1), a system aiming to reduce the planned downtime, by *performing an online upgrade*, and to remove the leading cause of upgrade failures—broken dependencies [1]—by presenting an alternative to tracking dependencies. While relying on the knowledge of the planned changes in data-formats in the new version, Imago treats the system-under-upgrade as a black box. We avoid breaking dependencies by installing the new version in a parallel universe—a logically distinct collection of resources, realized either using different hardware or through virtualization—and by transferring the persistent data, opportunistically, into the new version. Imago accesses the universe of the old version in a read-only manner, *isolating the production system from the upgrade operations*. When the data transfer is complete, Imago performs the switchover to the new version, *completing the end-to-end upgrade as an atomic operation.* Imago also enables the integration of long-running data conversions in an online upgrade and the live testing of the new version. §3
- We evaluate the benefits of Imago's mechanisms (*e.g.*, atomic upgrades, dependency isolation) through a systematic fault-injection approach, using our upgrade-centric fault model. Imago provides a better availability in the presence of upgrade faults than two alternative approaches, rolling upgrade and big flip [9] (result significant at the $p = 0.01$ level). §4

[1] The *imago* is the final stage of an insect or animal that undergoes a metamorphosis, *e.g.*, a butterfly after emerging from the chrysalis [8].

Compared with the existing strategies for online upgrades, Imago trades off the need for additional resources for an improved dependability of the online upgrade. While it cannot prevent latent configuration errors, Imago eliminates the internal single-points-of-failure for upgrade faults and the risk of breaking hidden dependencies by overwriting an existing system. Additionally, Imago avoids creating system states with mixed versions, which are difficult to test and to validate. Our results suggest that an *atomic, dependency-agnostic* approach, such as Imago, can improve the dependability of online software-upgrades despite hidden dependencies.

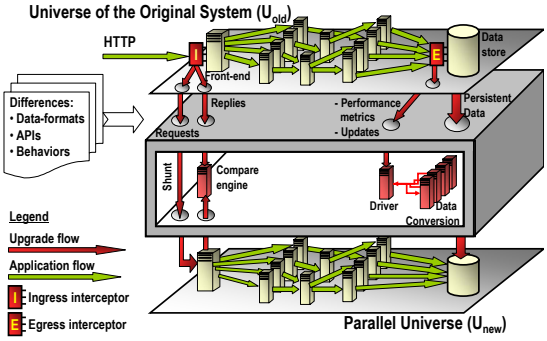


Fig. 1. Dependable upgrades with Imago.

2 Fault model for enterprise upgrades

Several classifications of upgrade faults have been proposed [10–13], but the fault categories are not disjoint, the criteria for establishing these categories remain unclear, or the classifications are relevant only for subsets of the upgrade faults. Moreover, data on upgrade-faults in the industry is scarce and hard to obtain due to the sensitivity of this subject. We analyze 55 upgrade faults from the best available sources, and, through statistical cluster-analysis, we establish four categories of upgrade faults.[2]

We combine data from three independent sources, which use different methodologies: a 2004 *user study* of system-administration tasks in an e-commerce system [12], a 2006 *survey* of database administrators [13], and a previously unpublished *field study* of bug reports filed in 2007 for the Apache web server [14]. While the administrators targeted by these studies focus on different problems and handle different workloads, we start from the premise that they use similar mental models during change-management tasks, which yield comparable faults. This hypothesis is supported by the observation that several faults have been reported in more than one study. Furthermore, as each of the three methodologies is likely to emphasize certain kinds of faults over others, combining these dissimilar data sets allows us to provide a better coverage of upgrade faults than previous studies.

2.1 The four types of upgrade faults

We conduct a *post-mortem* analysis of each fault from the three studies in order to determine its root cause [10]—*configuration* error, *procedural* error, *software* defect, *hardware* defect—and whether the fault has broken a hidden dependency, with repercussions for several components of the system-under-upgrade. Errors introduced while editing configuration files can can be further subdivided in three categories [11]: *typographical errors* (typos), *structural errors* (*e.g.* misplacing configuration directives), and *semantic errors* (*e.g.* ignoring constraints among configuration parameters). Additionally, a small number of configuration errors do not occur while editing configuration files (*e.g.*, setting incorrect access privileges). Operators can make procedural errors by performing an *incorrect action* or by violating the sequence of actions in the procedure through an *omission*, an *order inversion*, or the addition of a *spurious action*.

Most configuration and procedural errors break hidden dependencies (see Table 1). Incorrect or omitted actions sometimes occur because the operators ignore, or are not aware

[2] We discuss the statistical techniques in more detail in [14]. This technical report and the annotated fault data are available at `http://www.ece.cmu.edu/~tdumitra/upgrade_faults`.

Table 1. Examples of hidden dependencies (sorted by frequency).

Hidden dependency	Procedure violation	Impact
Service location: – File path – Network address	 Omission	Components unavailable, latent errors
Dynamic linking: – Library conflicts – Defective 3rd party components		Components unavailable
Database schema: – Application/database mismatch – Missing indexes	 Omission Omission	 Data unavailable Performance degradation
Access privileges to file system, database objects, or URLs: – Excessive – Insufficient – Unavailable (from directory service)	 Wrong action Omission Omission	 Vulnerability Components/data unavailable
Constraints among configuration parameters		Outage, degraded performance, vulnerability
Replication degree (*e.g.*, number of front-end servers online)	Omission, inversion, spurious action	Outage, degraded performance
Amount of storage space available	Omission	Transactions aborted
Client access to system-under-upgrade	Wrong action	Incorrect functionality
Cached data (*e.g.*, SSL certificates, DNS lookups, kernel buffer-cache)		Incorrect functionality
Listening ports	Omission	Components unavailable
Communication-protocol mismatch (*e.g.*, middle-tier not HTTP-compliant)		Components unavailable
Entropy for random-number generation		Deadlock
Request scheduling		Access denied unexpectedly
Disk speed	Wrong action	Performance degradation

of, certain dependencies among the system components (*e.g.*, the database schema queried by the application servers and the schema materialized in the production database). In 56% of cases, however, the operators break hidden dependencies (*e.g.*, by introducing shared-library conflicts) despite correctly following the mandated procedure. This illustrates the fact that even well-planned upgrades can fail because the complete set of dependencies is not always known in advance. We emphasize that the list of hidden dependencies from Table 1, obtained through a *post-mortem* analysis of upgrade faults, is not exhaustive and that other hidden dependencies might exist in distributed systems, posing a significant risk of failure for enterprise-system upgrades.

We perform statistical cluster-analysis, with five classification variables:[3] (i) the root cause of each fault; (ii) the hidden dependency that the fault breaks (where applicable); (iii) the fault location—*front-end*, *middle-tier*, or *back-end*—; (iv) the original classification, from the three studies; and (v) the cognitive level involved in the reported operator error. There are three cognitive levels at which humans solve problems and make mistakes [11]: the *skill-based* level, used for simple, repetitive tasks, the *rule-based* level, where problems are solved by pattern-matching, and the *knowledge-based* level, where tasks are approached by reasoning from first principles. The high-level fault descriptions from the three studies are sufficient for determining the values of the five classification

[3] We compare faults using the *Gower distance*, based on the categorical values of the classification variables. We perform agglomerative, hierarchical clustering with *average linkage* [15].

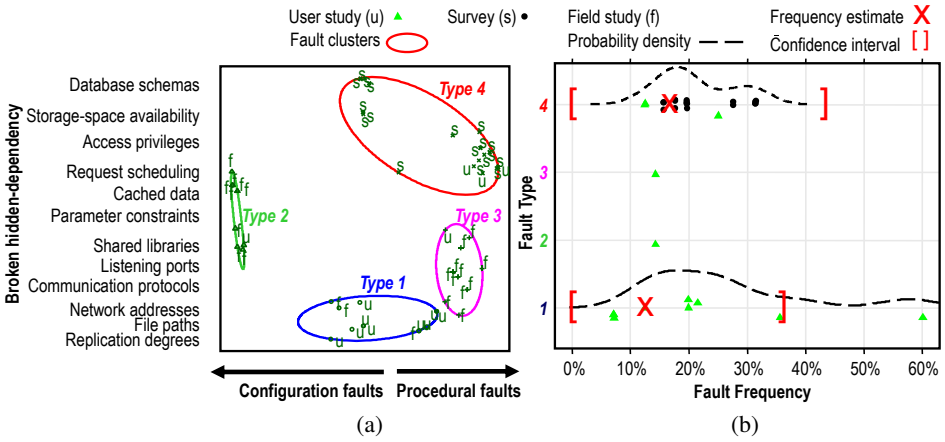


Fig. 2. Upgrade-centric fault model. Principal-component analysis (a) creates a two-dimensional shadow of the five classification variables, The survey and the user study also provide information about the distribution of fault-occurrence rates (b).

variables. We include all the faults reported in the three studies, except for software defects, faults that did not occur during upgrades and client-side faults. If a fault is reported in several sources, we include only one of its instances in the cluster analysis. We exclude software defects[4] from our taxonomy because they have been rigorously classified before [16] and because they are orthogonal to the upgrading concerns and might be exposed in other situations as well. Moreover, the survey from [1] suggests that most upgrade failures are not due to software defects.

This analysis suggests that there are four natural types of faults (Fig. 2):

- **Type 1** corresponds to simple configuration errors (typos or structural) and to procedural errors that occur on the skill-based cognitive level. These faults break dependencies on network addresses, file paths, or the replication degree.
- **Type 2** corresponds to semantic configuration errors, which occur on the knowledge-based cognitive level and which indicate a misunderstanding of the configuration directives used. These faults break dependencies on the request scheduling, cached data, or parameter constraints.
- **Type 3** corresponds to broken environmental dependencies, which are procedural errors that occur on the rule-based cognitive level. These faults break dependencies on shared libraries, listening ports, communication protocols, or access privileges.
- **Type 4** corresponds to data-access errors, which are complex procedural or configuration errors that occur mostly on the rule- and knowledge-based cognitive levels. These faults prevent the access to the system's persistent data, breaking dependencies on database schemas, access privileges, the replication degree, or the storage availability.

Faults that occur while editing configuration files are of type 1 or 2. Types 1–3 are located in the front-end and in the middle tier, and, except for a few faults due to omitted actions, they usually do not involve violating the mandated sequence of actions. Type 4 faults occur in the back-end, and they typically consist of wrong or out-of-sequence actions (except order inversions). Principal-component analysis (Fig. 2(a)) suggests that the four types of faults correspond to disjoint and compact clusters. Positive values on the x-axis indicate procedural faults, while negative values indicate faults that occur while editing

[4] The fault descriptions provided in the three studies allow us to distinguish the operator errors from the manifestations of software defects.

configuration files. The *y*-axis corresponds, approximately, to the hidden dependencies broken by the upgrade faults.

We also estimate how frequently these fault types occur during an upgrade (Fig. 2(b)), by considering the percentage of operators who induced the fault (during the user study) or the percentage of DBAs who consider the specific fault among the three most frequent problems that they have to address in their respective organizations (in the survey). We cannot derive frequency information from the field-study data. The individual estimations are imprecise, because the rate of upgrades is likely to vary among organizations and administrators, and because of the small sample sizes (5–51 subjects) used in these studies. We improve the precision of our estimations by combining the individual estimations for each fault type.[5] We estimate that Type 1 faults occur in 14.6% of upgrades (with a confidence interval of $[0\%, 38.0\%]$). Most Type 1 faults (recorded in the user study) occur in less than 21% of upgrades. Similarly, we estimate that Type 4 faults occur in 18.7% of upgrades (with a confidence interval of $[0\%, 45.1\%]$). Because faults of types 2 and 4 are predominantly reported in the field-study, we lack sufficient information to compute a statistically-significant fault frequency for these clusters.

Threats to validity. Each of the three studies has certain characteristics that might skew the results of the cluster analysis. Because the user study is concerned with the behavior of the operators, it does not report any software defects or hardware failures. Configuration errors submitted as bugs tend to be due to significant misunderstandings of the program semantics, and, as a result, our field study contains an unusually-high number of faults occurring on the knowledge cognitive level. Moreover, the results of bug searches are not repeatable because the status of bugs changes over time; in particular, more open bugs are likely to be marked as invalid or not fixed in the future. Finally, Crameri *et al.* [1], who identify broken dependencies as the leading cause of upgrade failures, caution that their survey is not statistically rigorous.

2.2 Tolerating upgrade faults

Several automated dependency-mining techniques have been proposed such as static and semantic analysis [18], but these approaches cannot provide a complete coverage of dependencies that only manifest dynamically, at runtime. Our upgrade-centric fault model emphasizes the fact that different techniques are required for tolerating each of the four types of faults. Modern software components check the syntax of their configuration files, and they are able to detect many Type 1 faults at startup (*e.g.*, syntax checkers catch 38%–83% of typos [11]). Type 2 faults are harder to detect automatically; Keller *et al.* [11] argue that checking the constraints among parameter values can improve the robustness against such semantic faults. To prevent faults that fall under Type 3, modern operating systems provide package managers that make a best-effort attempt to upgrade a software component along with all of its dependencies [19, 20]. Oliveira *et al.* propose validating the actions of database administrators using real workloads, which prevents some Type 4 faults but is difficult to implement when the administrator's goal is to change the database schema or the system's observable behavior.

Industry best-practices recommend carefully planning the upgrades and minimizing their risks by deploying the new version gradually, in successive stages [6]. For instance, two widely-used upgrading approaches are the *rolling upgrades* and the *big-flip* [9]. The first approach upgrades and then reboots each node, in a wave rolling through the cluster.

[5] The *precision* of a measurement indicates if the results are repeatable, with small variations, and the *accuracy* indicates if the measurement is free of bias. While in general it is not possible to improve the accuracy of the estimation without knowing the systematic bias introduced in an experiment, minimizing the sum of squared errors from dissimilar measurements improves the precision of the estimation [17].

The second approach upgrades half of the nodes while the other half continues to process requests, and then the two halves are switched. Both these approaches attempt to minimize the downtime by performing an *online upgrade*. A big flip has 50% capacity loss, but it enables the deployment of an incompatible system. Instead, a rolling upgrade imposes very little capacity loss, but it requires the old and new versions to interact with the data store and with each other in a compatible manner.

Commercial products for rolling upgrades provide no way of determining if the interactions between mixed versions are safe and leave these concerns to the application developers [7]. However, 47 of the 55 upgrade faults analyzed break dependencies that remain hidden from the developers or the operators performing the upgrade (see Table 1), and some procedural or configuration errors occur despite correctly following the upgrading procedure. This suggests that a novel approach is needed for improving the dependability of enterprise-system upgrades.

3 Design and implementation of Imago

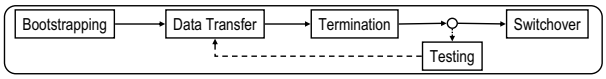
To provide dependable, online upgrades, we built Imago with three design goals:

- **Isolation**: The dependencies within the old system must be isolated from the upgrade operations.
- **Atomicity**: At any time, the clients of the system-under-upgrade must access the full functionality of either the old or the new systems, but not both. The end-to-end upgrade must be an atomic operation.
- **Fidelity**: The testing environment must reproduce realistically the conditions of the production environment.

Distributed enterprise-systems typically have one or more *ingress points* (**I**), where clients direct their requests, and one or more *egress points* (**E**), where the persistent data is stored (see Fig. 1). The remainder of the infrastructure (*i.e.*, the request paths between **I** and **E**) implements the business-logic and maintains only volatile data, such as user-sessions or cached data-items. We install the new system in a *parallel universe*—a logically distinct collection of resources, including CPUs, disks, network links, *etc*.—that is isolated from the universe where the old system continues to operate. The new system may be a more recent version of the old system, or it may be a completely different system that provides similar or equivalent functionality. Imago updates the persistent data of the new system through an opportunistic data-transfer mechanism. The logical isolation between the universe of the old system, $\mathbf{U_{old}}$, and the universe of the new system, $\mathbf{U_{new}}$, ensures that the two parallel universes do not share resources and that the upgrade process, operating on $\mathbf{U_{new}}$, has no impact on the dependencies encapsulated in $\mathbf{U_{old}}$. Our proof-of-concept implementation provides isolation by using separate hardware resources, but similar isolation properties could be achieved through virtualization. Because Imago always performs read-only accesses on $\mathbf{U_{old}}$, the dependencies of the old system cannot be broken and need not be known in advance.

Assumptions. We make three assumptions. We assume that (1) the system-under-upgrade has well-defined, static ingress and egress points; this assumption simplifies the task of monitoring the request-flow through $\mathbf{U_{old}}$ and the switchover to $\mathbf{U_{new}}$. We further assume that (2) the workload is dominated by read-only requests; this assumption is needed for guaranteeing the eventual termination of the opportunistic data-transfer. Finally, we assume that the system-under-upgrade provides hooks for: (3a) flushing in-progress updates (needed before switchover); and (3b) reading from $\mathbf{U_{old}}$'s data-store without locking objects or obstructing the live requests in any other way (to avoid interfering with the live workload). We do not assume any knowledge of the internal communication paths between the ingress and egress points.

These assumptions define the class of distributed systems that can be upgraded using Imago. For example, enterprise systems with three-tier architectures—composed of a front-end tier that manages client connections, a middle tier that implements the business logic of the application, and a back-end tier where the persistent data is stored—satisfy these assumptions. An ingress point typically corresponds to a front-end proxy or a load-balancer, and an egress point corresponds to a master database in the back-end. E-commerce web sites usually have read-mostly workloads [21], satisfying the second assumption. The two $\mathbf{U_{old}}$ hooks required in the third assumption are also common in enterprise systems; for instance, most application servers will flush the in-progress updates to their associated persistent storage before shutdown, and most modern databases support snapshot isolation[6] as an alternative to locking.



Upgrade procedure. Imago uses a procedure with five phases: bootstrapping, data-transfer, termination, testing, and switchover. Imago lazily transfers the persistent data from the system in $\mathbf{U_{old}}$ to the system in $\mathbf{U_{new}}$, converts it into the new format, monitors the data-updates reaching $\mathbf{U_{old}}$'s egress points and identifies the data objects that need to be re-transferred in order to prevent data-staleness. The live workload of the system-under-upgrade, which accesses $\mathbf{U_{old}}$'s data store concurrently with the data-transfer process, can continue to update the persistent data. The egress interceptor, **E**, monitors $\mathbf{U_{old}}$'s data-store activity to ensure that all of the updated or new data objects are eventually (re)-transferred to $\mathbf{U_{new}}$. Because Imago always performs read-only accesses on $\mathbf{U_{old}}$, the dependencies of the old system cannot be broken and need not be known in advance. Moreover, **E** monitors the load and the performance of $\mathbf{U_{old}}$'s data store, allowing Imago to regulate its data-transfer rate in order to avoid interfering with the live workload and satisfying our isolation design-goal. This upgrade procedure is described in detail in [22].

The most challenging aspect of an online upgrade is the switchover to the new version. The data transfer will eventually terminate if the transfer rate exceeds the rate at which $\mathbf{U_{old}}$'s data is updated (this is easily achieved for read-mostly workloads). To complete the transfer of the remaining in-progress updates, we must enforce a brief period of quiescence for $\mathbf{U_{old}}$. Imago can enforce quiescence using the **E** interceptor, by marking all the database tables read-only, or using the **I** interceptors, by blocking all the incoming write requests. The first option is straightforward: the database prevents the system in $\mathbf{U_{old}}$ from updating the persistent state, allowing the data-transfer to terminate. This approach is commonly used in the industry due to its simplicity [7].

If the system-under-upgrade can not tolerate the sudden loss of write-access to the database, Imago can instruct the **I** interceptors to block all the requests that might update $\mathbf{U_{old}}$'s persistent data (read-only requests are allowed to proceed). In this case, Imago must flush the in-progress requests to $\mathbf{U_{old}}$'s data store in order to complete the transfer to $\mathbf{U_{new}}$. Imago does not monitor the business logic of $\mathbf{U_{old}}$, but the **I** interceptors record the active connections of the corresponding ingress servers to application servers in the middle tier and invoke the flush-hooks of these application servers. When all the interceptors report the completion of the flush operations, the states of the old and new systems are synchronized, and Imago can complete the switchover by redirecting all the traffic to $\mathbf{U_{new}}$ (this protocol is described in Fig. 3). The volatile data (*e.g.*, the user sessions) is not transferred to $\mathbf{U_{new}}$ and is reinitialized after switching to the new system. Until this phase the progress of the ongoing upgrade is transparent to the clients, but after the switchover only the new version will be available.

[6] This mechanism relies on the multi-versioning of database tables to query a snapshot of the database that only reflects committed transactions and is not involved in subsequent updates.

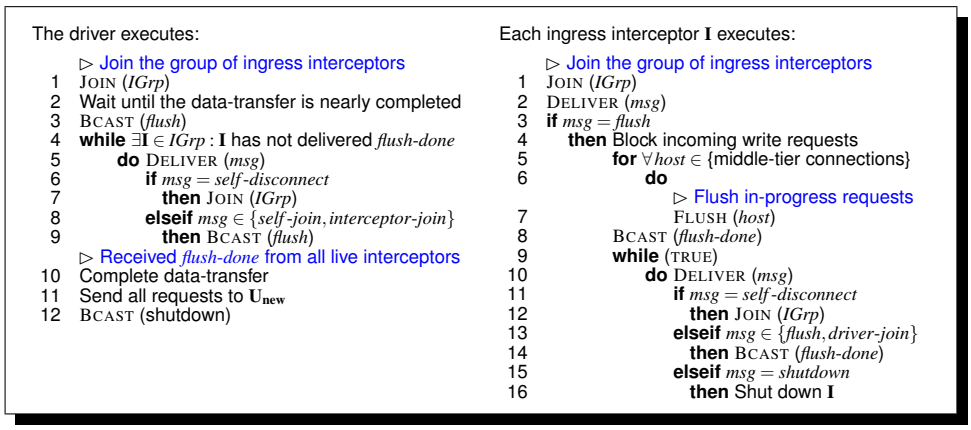


Fig. 3. Pseudocode of the switchover protocol.

Imago also supports a series of iterative testing phases before the switchover. Imago checkpoints the state of the system in $\mathbf{U_{new}}$ and then performs *offline testing*—using pre-recorded or synthetically-generated traces that check the coverage of all of the expected features and behaviors—and *online testing*—using the live requests recorded at **I**. In the second case, the testing environment is nearly identical to the production environment, which satisfies our fidelity design-goal. Quiescence is not enforced during the testing phase, and the system in $\mathbf{U_{old}}$ resumes normal operation while **E** continues to monitor the persistent-state updates. At the end of this phase, Imago rolls the state of the system in $\mathbf{U_{new}}$ back to the previous checkpoint, and the data transfer resumes in order to account for any updates that might have been missed while testing. A detailed discussion of the testing phase is beyond the scope of this paper.

After adequate testing, the upgrade can be rolled back, by simply discarding the $\mathbf{U_{new}}$ universe, or committed, by making $\mathbf{U_{new}}$ the production system, satisfying our atomicity design-goal. Imago treats the system-under-upgrade as a black box. Because we do not rely on any knowledge of the internal communication paths between the ingress and egress points of $\mathbf{U_{old}}$ and because all of the changes required by the upgrade are made into $\mathbf{U_{new}}$, Imago does not break any hidden dependencies in $\mathbf{U_{old}}$.

Implementation. Imago has four components (see Fig. 1): the *upgrade driver*, which transfers data items from the data store of $\mathbf{U_{old}}$ to that of $\mathbf{U_{new}}$ and coordinates the upgrade protocol, the *compare-engine*, which checks the outputs of $\mathbf{U_{old}}$ and $\mathbf{U_{new}}$ during the testing phase, and the **I** and **E** interceptors. The upgrade driver is a process that executes on hardware located outside of the $\mathbf{U_{old}}$ and $\mathbf{U_{new}}$ universes, while **I** and **E** are associated with the ingress and egress points of $\mathbf{U_{old}}$. We implement the **E** interceptor by monitoring the query log of the database. The **I** interceptor uses library interposition to redefine five system calls used by the front-end web servers: `accept()` and `close()`, which mark the life span of a client connection, `connect()`, which opens a connection to the middle tier, and `read()` and `writev()`, which reveal the content of the requests and replies, respectively. These five system calls are sufficient for implementing the functionality of the **I** interceptor. We maintain a memory pool inside the interceptor, and the redefined `read()` and `writev()` system-calls copy the content of the requests and replies into buffers from this memory pool. The buffers are subsequently processed by separate threads in order to minimize the performance overhead.

In order to complete the data transfer, the upgrade driver invokes the switchover protocol from Fig. 3. We use reliable group-communication primitives to determine when

all the interceptors are ready to switch: JOIN allows a process to join the group of interceptors and to receive notifications when processes join or disconnect from the group; BCAST reliably sends a message to the entire group; and DELIVER delivers messages in the same order at all the processes in the group. These primitives are provided by the Spread package [23]. The switchover protocol also relies on a FLUSH operation, which flushes the in-progress requests from a middle-tier server. Each **I** interceptor invokes the FLUSH operation on the application servers that it has communicated with.

We have implemented the FLUSH operation for the Apache and JBoss servers. For Apache, we restart the server with the `graceful` swirch, allowing the current connections to complete. For JBoss, we change the timestamp of the web-application archive (the `application.war` file), which triggers a redeployment of the application. Both these mechanisms cause the application servers to evict all the relevant data from their caches and to send the in-progress requests to the back-end. This switchover protocol provides strong consistency, and it tolerates crashes and restarts of the driver or the interceptors. All the modules of Imago are implemented in C++ (see Table 2). The application bindings contain all the application-specific routines (*e.g.*, data conversion) and constitute 14% of the code. Most of this application-specific code would also be necessary to implement and offline upgrade.

Table 2. Structure of Imago's code.

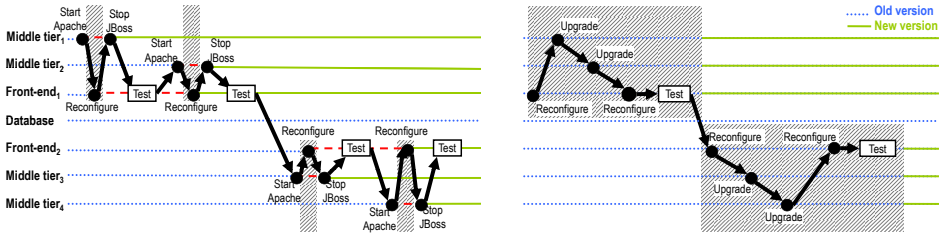
	Lines of code	Size in memory
Upgrade driver	2,038	216 kB
Egress interceptor	290	
Ingress interceptor	2,056	228 kB
Switchover library	1,464	
Compare engine	571	48 kB
Common libraries	591	44 kB
Application bindings	1,113	108 kB
Total	8,123	—

4 Experimental evaluation

We evaluate the dependability of enterprise-system upgrades performed using Imago. Specifically, we seek answers to the following questions:

- What overhead does Imago impose during a successful upgrade? §4.1
- Does Imago improve the availability in the presence of upgrade faults? §4.2
- How do types 1–4 of upgrade faults affect the reliability of the upgrade? §4.3

Upgrade scenario. We use Imago to perform an upgrade of RUBiS (the Rice University Bidding System) [24], an open-source online bidding system, modeled after eBay. RUBiS has been studied extensively, and several of its misconfiguration- and failure-modes have been previously reported [12, 13]. RUBiS has multiple implementations (*e.g.*, using PHP, EJB, Java Servlets) that provide the same functionality and that use the same data schema. We study an upgrade scenario whose goal is to upgrade RUBiS from the version using Enterprise Java Beans (EJB) to the version implemented in PHP. The system-under-upgrade is a three-tier infrastructure, comprising a front-end with two Apache web servers, a middle tier with four Apache servers that execute the business logic of RUBiS, and a MySQL database in the back-end. More specifically, the upgrade aims to replace the JBoss servers in the middle tier with four Apache servers where we deploy the PHP scripts that implement RUBiS's functionality. The RUBiS database contains 8.5 million data objects, including 1 million items for sale and 5 million bids. We use two standard workloads, based on the TPC-W specification [21], which are typical for e-commerce web sites. The performance bottleneck in this system is the amount of physical memory in the front-end web servers, which limits the system's capacity to 100 simultaneous clients. We conduct our experiments in a cluster with 10 machines (Pentium 4 at 2.4 GHz, 512 MB RAM), connected by a 100 Mbps LAN.



(a) Rolling upgrade (b) Big-flip upgrade

Fig. 4. Current approaches for online upgrades in RUBiS.

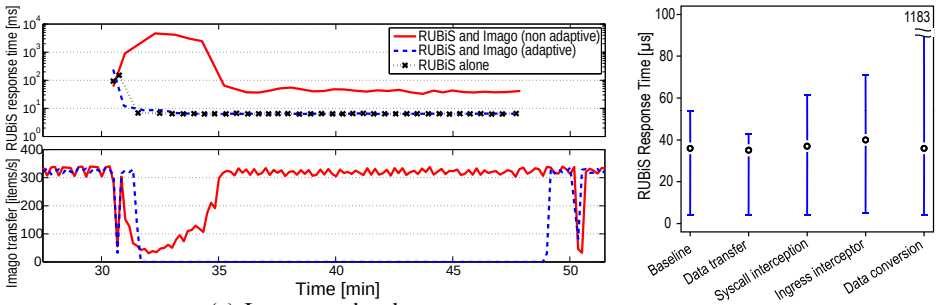
We compare Imago with two alternative approaches, rolling upgrades and big flip (see Section 2.2). These procedures are illustrated in Fig. 4. In both cases, the front-end and back-end remain shared between the old and new versions. Rolling upgrades run for a while in a mode with mixed versions, with a combination of PHP (Apache) and EJB (JBoss) nodes in the middle tier, while the big flip avoids this situation but uses only half of the middle-tier servers. With the former approach an upgraded node is tested online (Fig. 4(a)), while the latter approach performs offline tests on the upgraded nodes and re-integrates them in the online system only after the flip has occurred (Fig. 4(b)). In contrast, Imago duplicates the entire architecture, transferring all of the 8.5 million RUBiS data-items to $\mathbf{U_{new}}$, in order to avoid breaking dependencies during the upgrade.

Methodology. We estimate Imago's effectiveness in performing an online upgrade, in the absence of upgrade-faults, by comparing the client-side latency of RUBiS before, and during, the upgrade. We assess the impact of broken dependencies by injecting upgrade faults, according to the fault model presented in Section 2, and by measuring the effect of these faults on the system's expected availability. Specifically, we estimate the system's *yield* [9], which is a fine-grained measure of availability with a consistent significance for windows of peak and off-peak load:

$$\text{Yield}(fault) = \frac{\text{Requests}_{completed}(fault)}{\text{Requests}_{issued}}$$

We select 12 faults (three for each fault type) from the data analyzed in Section 2, prioritizing faults that have been confirmed independently, in different sources or in separate experiments from the same source. We repeat each fault-injection procedure three times and we report the average impact, in terms of response time and yield-loss, on the system. Because this manual procedure limits us to injecting a small number of faults, we validate the results using statistical-significance tests, and we complement these experiments with an automated injection of Type 1 faults.

From a client's perspective, the upgrade faults might cause a full outage, a partial outage (characterized by a higher response time or a reduced throughput), a delayed outage (due to latent errors) or they might have no effect at all. A full outage ($Yield = 0$) is recorded when the upgrade-fault immediately causes the throughput of RUBiS to drop to zero. Latent errors remain undetected until they are eventually exposed by external factors (*e.g.*, a peak load) or by system-configuration changes. To be conservative in our evaluation, we consider that (i) the effect of a latent error is the same as the effect of a full outage ($Yield = 0$); (ii) an upgrade can be stopped as soon as a problem is identified; and (iii) all errors (*e.g.*, HTTP-level or application-level errors) are detected. An upgrading mechanism is able to mask a dependency-fault when the fault is detected before reintegrating the affected node in the online system. To avoid additional approximations, we



(a) Imago overhead. (b) Overhead breakdown.

Fig. 5. Upgrade overhead on a live RUBiS system.

do not attempt to estimate the durations of outages caused by the broken dependencies. As the yield calculations do not include the time needed to mitigate the failures, the values reported estimate the initial impact of a fault but not the effects of extended outages. While the result that Imago provides better availability under upgrade faults is statistically significant, the *quantitative* improvements depend on the system architecture and on the specific faults injected, and they might not be reproducible for a different system-under-upgrade. The goal of our fault-injection experiments is to determine the *qualitative* reasons for unavailability during online upgrades, and to emphasize the opportunities for improving the current state-of-the-art.

4.1 Performance overhead without faults

The latency of querying the content of a data item from $\mathbf{U_{old}}$ and inserting it in $\mathbf{U_{new}}$ dominates the performance of the data-transfer; less than 0.4% out of the 5 ms needed, on average, to transfer one item are spent executing Imago's code. Fig. 5(a) shows the impact of the data transfer on RUBiS's end-to-end latency (measured at the client-side). If requests arrive while a data-transfer is in progress, the response times increase by three orders of magnitude (note the log scale in the top panel of Fig. 5(a)). These high latencies correspond to a sharp drop in the transfer rate as the $\mathbf{U_{old}}$ database tries to adjust to the new load. However, Imago can use the information collected by the **E** interceptor to self-regulate in order to avoid overloading the production system. We have found that the incoming query rate for $\mathbf{U_{old}}$'s database provides sufficient warning: if Imago uses a simple adaptation policy, which pauses the data transfer when the RUBiS clients issue more than 5 queries/s, the end-to-end latency is indistinguishable from the case when clients do not compete with Imago for $\mathbf{U_{old}}$'s resources (Fig. 5(a)). After resuming the data transfer, Imago must take into account the data items added by RUBiS's workload. These new items will be transferred during subsequent periods of client inactivity. Under a scenario with 1000 concurrent clients, when the site is severely overloaded, Imago must make progress, opportunistically, for 2 minutes per hour in order to catch up eventually and complete the data transfer.

Fig. 5(b) compares the overheads introduced by different Imago components (the error bars indicate the 90% confidence intervals for the RUBiS response time). The **I** interceptors impose a fixed overhead of 4 ms per request; this additional processing time does not depend on the requests received by the RUBiS front-ends. When Imago performs a data conversion (implemented by modifying the RUBiS code, in order to perform a database-schema change during the upgrade), the median RUBiS latency is not affected but the maximum latency increases significantly. This is due to the fact that the simple adaptation policy described above is not tuned for the data-conversion scenario.

Table 3. Description of upgrade-faults injected.

	Name / Instances [source]	Location	Fault-Injection Procedure	Local Manifestation
Type 1	`wrong_apache` 2 [12]	Front-end	Restarted wrong version of Apache on one front-end.	Server does not forward requests to the middle tier.
	`config_nochange` 1 [12]	Front-end	Did not reconfigure front-end after middle-tier upgrade.	Server does not forward requests to the middle tier.
	`config_staticpath` 2 [12, 14]	Front-end	Mis-configured path to static web pages on one front-end.	Server does not forward requests to the middle tier.
Type 2	`config_samename` 1 [12]	Front-end	Configured identical names for the application servers.	Server communicates with a single middle-tier node.
	`apache_satisfy` 1 [14]	Middle tier	Used `Satisfy` directive incorrectly.	Clients gain access to restricted location.
	`apache_largefile` 2 [14]	Middle tier	Used `mmap()` and `sendfile()` with network file-system.	No negative effect (could not replicate the bug).
Type 3	`apache_lib` 1 [14]	Middle tier	Shared-library conflict.	Cannot start application server.
	`apache_port_f` 1 [14]	Front-end	Listening port already in use by another application.	Cannot start front-end web server.
	`apache_port_m` 1 [14]	Middle tier	Listening port already in use by another application.	Cannot start application sever.
Type 4	`wrong_privileges` 2 [12, 13]	Back-end	Wrong privileges for RUBiS database user.	Database inaccessible to the application servers.
	`wrong_shutdown` 2 [12, 13]	Back-end	Unnecessarily shut down the database.	Database inaccessible to the application servers.
	`db_schema` 4 [13]	Back-end	Changed DB schema (renamed `bids` table).	Database partially inaccessible to application servers.

The rolling upgrade does not impose any overhead, because sequentially rebooting all the middle-tier nodes does not affect the system's latency or throughput. The big flip imposes a similar run-time overhead as Imago because half of the system is unavailable during the upgrade. With Imago, the upgrade completes after ≈13h, which is the time needed for transferring all the persistent data plus the time when access to $\mathbf{U_{old}}$ was yielded to the live workload. This duration is comparable to the time required to perform an offline upgrade: in practice, typical Oracle and SAP migrations require planned downtimes of tens of hours to several days [25].

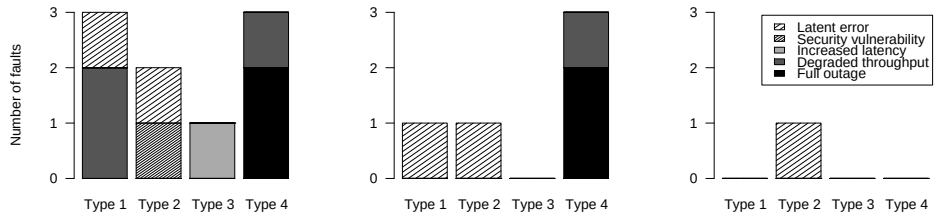
Before switching to $\mathbf{U_{new}}$, Imago enforces quiescence by either marking the database tables read-only, or by rejecting write requests at the **I** interceptors and flushing the in-progress updates to the persistent storage. When the middle-tier nodes are running Apache/PHP servers, the flush operation takes 39 s on average, including the synchronization required by the protocol from Fig. 3. In contrast, flushing JBoss application servers requires only 4.4 s on average, because in this case we do not need to restart the entire server. The switchover mechanism does not cause a full outage, as the clients can invoke the read-only functionality of RUBiS (*e.g.*, searching for items on sale) while Imago is flushing the in-progress requests. Moreover, assuming that the inter-arrival times follow an exponential distribution and the workload mix includes 15% write requests (as specified by TPC-W [21]), we can estimate the maximum request rate that the clients may issue without being denied access. If the switchover is performed during a time window when the live request rate does not exceed 0.5 requests/min, the clients are unlikely (p=0.05) to be affected by the flush operations.

4.2 Availability under upgrade-faults

Table 3 describes the upgrade-faults injected and their immediate, local manifestation. We were unable to replicate the effects of one fault (`apache_largefile`, which was reported as bugs 42751 and 43232 in the field study) in our experimental test-bed. We inject the remaining 11 faults in the front-end (5 faults), middle tier (4 faults) and the back-end (3 faults) during the online upgrade of RUBiS. In a rolling upgrade, a node is reintegrated after the local upgrade, and resulting errors might be propagated to the client. The big flip can mask the upgrade-faults in the offline half but not in the shared database. Imago masks all the faults that can be detected (*i.e.*, those that do not cause latent errors).

Fig.6 shows the impacts that Types 1–4 of upgrade faults have on the system-under-upgrade. Certain dependency-faults lead to an increase in the system's response time. For instance, the `apache_port_f` fault doubles the connection load on the remaining front-end server, which leads to an increased queuing time for the client requests and a 8.3% increase in response-time when the fault occurs. This outcome is expected during a big-flip, but not during a rolling upgrade (see Fig. 4). This fault does not affect the system's throughput or yield because all of the requests are eventually processed and no errors are reported to the clients.

The `config_nochange` and `wrong_apache` faults prevent one front-end server from connecting to the new application servers in the middle tier. The front-end server affected continues to run and to receive half of the client requests, but it generates HTTP errors ($Yield = 0.5$). Application errors do not manifest themselves as noticeable degradations of the throughput, in terms of the rate of valid HTTP replies, measured at either the client-side or the server-side. These application errors can be detected only by examining the actual payload of the front-end's replies to the client's requests. For instance, `db_schema` causes intermittent application errors that come from all four middle-tier nodes. As this fault occurs in the back-end, both the rolling upgrade and the big flip are affected. Imago masks this fault because it does not perform any configuration actions on $\mathbf{U_{old}}$. Similarly, Imago is the only mechanism that masks the remaining Type 4, `wrong_privileges` and `wrong_shutdown`. The `apache_satisfy` fault leads to a potential security vulnerability, but does not affect the yield or the response time. This fault can be detected, by issuing requests for the restricted location, unlike the `config_staticpath` fault, which causes the front-end to serve static web pages from a location that might be removed in the future. Because this fault does not have any observable impact during the rolling upgrade or the big flip, we consider that it produces a latent error. Imago masks `config_staticpath` because the obsolete location does not exist in $\mathbf{U_{new}}$, and the fault becomes detectable. The `config_samename` fault prevents one front-end server from forwarding requests to one middle-tier node, but the three application servers remaining can successfully handle the RUBiS workload, which is not computationally-intensive. This fault produces a latent error that might be exposed by future changes in the workload or the system architecture and is the only fault that Imago is not able to mask.



(a) Rolling Upgrade. (b) Big Flip. (c) Imago.

Fig. 6. Impact of upgrade faults.

The rolling upgrade masks 2 faults, which occur in the middle tier and do not degrade the response time or the yield, but have a visible manifestation (the application server fails to start). The big flip masks 6 faults that are detected before the switch of the halves. Imago masks 10 out of the 11 injected faults, including the ones masked by the big flip, and excluding the latent error. A paired, one-tailed t-test[7] indicates that, under upgrade faults, Imago provides a better yield than the rolling upgrade (significant at the $p = 0.01$ level) and than the big flip (significant at the $p = 0.05$ level).

4.3 Upgrade reliability

We observe in Fig.6 that broken environmental dependencies (Type 3) have only a small impact on enterprise-system upgrades, because their manifestations (*e.g.*, a server's failure to start) are easy to detect and compensate for in any upgrading mechanism. Rolling upgrades create system states with mixed versions, where hidden dependencies can be broken. Contrary to the conventional wisdom, these faults can have a global impact on the system-under-upgrade, inducing outages, throughput- or latency-degradations, security vulnerabilities or latent errors.

Compared with a big flip, Imago improves the availability because (i) it removes the *single points of failure* for upgrade faults and (ii) it performs a clean installation of the new system. For instance, the `config_staticpath` fault induces a latent error during the big flip because the upgrade overwrites an existing system. The database represents a single point of failure for the big flip, and any Type 4 fault leads to an upgrade failure for this approach. Such faults do not always cause a full outage; for instance, the `db_schema` fault introduces a throughput degradation (with application errors). However, although in this case the application error-rate is relatively low (9% of all replies), the real impact is much more severe: while clients can browse the entire site, they cannot bid on any items. In contrast, Imago eliminates the single-points-of-failure for upgrade faults by avoiding an in-place upgrade and by isolating the system version in $\mathbf{U_{old}}$ from the upgrade operations.

Imago is vulnerable to latent configuration errors such as `config_samename`, which escapes detection. This failure is not the result of breaking a shared dependency, but corresponds to an incorrect invariant of the new system, established during a fresh install. This emphasizes the fact that any upgrading approach, even Imago, will succeed only if an effective mechanism for testing the upgraded system is available.

Because our qualitative evaluation does not suggest how often the upgrade faults produce latent errors, we inject Type 1 faults automatically, using ConfErr [11]. ConfErr explores the space of likely configuration errors by injecting one-letter omissions, insertions, substitutions, case alterations and transpositions that can be created by an operator who mistakenly presses keys in close proximity to the mutated character. We randomly inject 10 typographical and structural faults into the configuration files of Apache web servers from the front-end and the middle tier, focusing on faults that are likely to occur during the upgrade (*i.e.*, faults affecting the configuration directives of `mod_proxy` and `mod_proxy_balancer` on the front-end and of `mod_php` on the middle tier). Apache's syntactic analyzer prevents the server from starting for 5 front-end and 9 middle-tier faults. Apache starts with a corrupted address or port of the application server after 2 front-end faults and with mis-configured access privileges to the RUBiS URLs after 1 middle-tier fault. The remaining three faults, injected in the front-end, are benign because they change a parameter (the route from a `BalancerMember` directive) that must be unique but that has no constraints on other configuration settings. These faults might have introduced latent errors if the random mutation had produced identical routes for two

[7] The t-test takes into account the pairwise differences between the yield of two upgrading approaches and computes the probability p that the *null hypothesis*—that Imago doesn't improve the yield—is true [17].

application servers; however, the automated fault-injection did not produce any latent errors. This suggests that latent errors are uncommon and that broken dependencies, which are tolerated by Imago, represent the predominant impact of Type 1 faults.

5 Lessons learned

Offline upgrades of critical enterprise-systems (*e.g.*, banking infrastructures) provide the opportunity for performing extensive tests for accepting or rejecting the outcome of the upgrade. Online upgrades do not have this opportunity; when there are mixed versions, system states are often short-lived and cannot be tested adequately, while the system-under-upgrade must recover quickly from any upgrade faults. Unlike the existing strategies for online upgrade, which rely on tracking dependencies, Imago trades off spatial overhead (*i.e.*, additional hardware and storage space) for an increased dependability of the online upgrade. Imago was designed for upgrading enterprise systems with traditional three-tier architectures. The current implementation cannot be readily applied to certain kinds of distributed systems, such as peer-to-peer systems, which violate the first assumption by accommodating large numbers of dynamically added ingress-points, or data-intensive computing (*e.g.*, MapReduce), which distribute their persistent data throughout the infrastructure and do not have a well-defined egress point. However, the availability improvements derive from the three properties (isolation, atomicity and fidelity) that Imago provides. Specifically, the isolation between the old and new versions reduces the risk of breaking hidden dependencies, which is the leading cause of upgrade failure [1], while performing the end-to-end upgrade as an atomic operation increases the upgrade reliability by avoiding system states with mixed versions. Imago improves the upgrade dependability because it implements *dependency-agnostic upgrades*. In the future, we plan to investigate mechanisms for implementing the isolation, atomicity and fidelity properties in other distributed-system architectures, and for reducing Imago's spatial overhead through virtualization.

Moreover, upgrades that aim to integrate several enterprise systems (*e.g.*, following commercial mergers and acquisitions) require complex data conversions for changing the data schema or the data store, and such data conversions are often tested and deployed in different environments [13], which increases the risk of upgrade failure. Imago is able to integrate complex data-conversions in an online upgrade and to test the new version online, in an environment nearly identical to the deployment system. While an in-depth discussion of these topics is outside the scope of this paper, we note that there are two major design choices for software-upgrade mechanisms: (i) whether the upgrade will be performed *in-place*, replacing the existing system, and (ii) whether the upgrade mechanisms will allow *mixed versions*, which interact and synchronize their states until the old version is retired. Table 4 compares these choices. Mixed versions save storage space because the upgrade is concerned with only the parts of the data schema that change between versions. However, mixed versions present the risk of breaking hidden dependencies; *e.g.*, if the new version includes a software defect that corrupts the persistent data, this corruption will be propagated back into the old version, replacing the master copy. Mixed, interacting versions also require an indirection layer, for dispatching requests to the appropriate version [26], which might introduce run-time overhead and will likely impose downtime when it is first installed. A system without mixed versions performs the upgrade in a single direction, from the old version to the new one. However, for in-place upgrades, the overhead due to data conversions can have a negative impact on the live workload. When, instead, an upgrade uses separate resources for the new version, the computationally-intensive processing can be performed downstream, on the target nodes (as in the case of Imago). As we have shown in Section 4, in-place upgrades introduce a high risk of breaking hidden dependencies, which degrades the expected availability.

The most significant disadvantage of out-of-place upgrades is the spatial overhead imposed. However, the cost of new hardware decreases while unavailability becomes more

Table 4. Design choices for online upgrades in enterprise systems.

	In-Place	Out-of-Place

Mixed Versions	– Risk propagating corrupted data
	– Need indirection layer, with:
	  – Potential run-time overhead
	  – Installation downtime
	– Incur run-time overhead for data conversions
	– Risk breaking hidden dependencies
		– Risk propagating corrupted data
		– Need indirection layer, with:
		  – Potential run-time overhead
		  – Installation downtime
		– Incur spatial overhead

Atomic	– Incur run-time overhead for data conversions – Risk breaking hidden dependencies – Incur spatial overhead	– Incur spatial overhead

expensive [9], and enterprises sometimes take advantage of a software upgrade to renew their hardware as well [25, 27]. Moreover, Imago requires additional resources only for implementing and testing the online upgrade, and storage and compute cycles could be leased, for the duration of the upgrade, from existing cloud-computing infrastructures (*e.g.*, the Amazon Web Services). This suggests that Imago is the first step toward an *upgrades-as-a-service* model, making complex upgrades practical for a wide range of enterprise systems.

6 Related Work

In our previous work [22], we have outlined the upgrade procedure on which Imago is based. Here, we review the research related to our contributions in this paper.

6.1 Upgrade Fault-Models

Oppenheimer *et al.* [10] study 100+ *post-mortem* reports of user-visible failures from three large-scale Internet services. They classify failures by location[8] (front-end, back-end and network) and by the root cause of the failure[8] (operator error, software fault, hardware fault). Most failures reported occurred during change-management tasks, such as scaling or replacing nodes and deploying or upgrading software. Nagaraja *et al.* [12] report the results of a user study[9] with 21 operators and observe seven classes of faults:[9] global misconfiguration, local misconfiguration, start of wrong software version, unnecessary restart of software component, incorrect restart, unnecessary hardware replacement, wrong choice of hardware component. Oliveira *et al.* [13] present a survey of 51 database administrators,[9] who report eight classes of faults:[9] deployment, performance, general-structure, DBMS, access-privilege, space, general-maintenance, and hardware. Keller *et al.* [11] study configuration errors and classify them according to their relationship with the format of the configuration file[9] (typographical, structural or semantic) and to the cognitive level where they occur[9] (skill, rule or knowledge). These models do not constitute a rigorous taxonomy of upgrade faults. Some classifications are too coarse-grained [10] or relevant for only a subset of the upgrade faults [11]. In many cases, the fault categories are not disjoint and the criteria for establishing these categories are not clearly stated.

6.2 Online Upgrades

The problem of dynamic software update (DSU), *i.e.*, modifying a running program on-the-fly, has been studied for over 30 years. Perhaps the most advanced DSU techniques are implemented in the Ginseng system, of Neamtiu *et al.* [28], which uses static analysis to ensure the safety and timeliness of updates (*e.g.*, establishing constraints to prevent old code from accessing new data) and supports all the changes required for updating several practical systems. When upgrading distributed systems with replicated components

[8] We use this subdivision as a classification variable in our upgrade fault-model (Section 2).

[9] We use this data to develop our upgrade fault-model (Section 2).

(*e.g.*, multiple application servers in the middle tier), practitioners often prefer rolling upgrades [9], because of their simplicity. DSU techniques are difficult to use in practice-because they require programmers to annotate (*e.g.*, indicating suitable locations for performing the update) or to modify the source code of the old and new versions. Moreover, active code (*i.e.*, functions on the call stack of the running program) cannot be replaced, and updating multi-threaded programs remains a challenging task [29]. Like Imago, DSU techniques require state conversion between program versions [28], but Imago never produces mixed versions and does not have to establish correctness conditions for the interactions among these versions. Imago performs the entire end-to-end upgrade as one atomic action.

6.3 Dependable Upgrades

To improve the dependability of single-node upgrades, modern operating systems include package-management tools, which track the dependencies among system components in depth, to prevent broken dependencies. Instead of tracking the dependencies of each package, Crameri *et al.* [1] suggest that the risk of upgrade failure can be reduced by testing new or updated packages in a wide variety of user environments and by staging the deployment of upgrades to increasingly dissimilar environments. Imago is closest in spirit to the previous upgrading approaches that avoid dependency tracking by isolating the new version from the old one. Lowell *et al.* [30] propose upgrading operating systems in a separate, lightweight virtual-machine and describe the Microvisor virtual-machine monitor, which allows a full, "devirtualized" access to the physical hardware during normal operation. The online applications are migrated to a separate virtual machine during the upgrade. To facilitate this application-migration process, Potter *et al.* [31] propose AutoPod, which virtualizes the OS's system calls, allowing applications to migrate among location-independent "pods". These approaches do not provide support for application upgrades. While providing better isolation properties than other in-place upgrades, the approaches based on virtual machines induce run-time overhead, which might break dependencies on performance levels (*e.g.*, applications that disable write-access when the response time increases).

Multi-node upgrades are vulnerable to Types 1–4 of upgrade faults. Nagaraja *et al.* [12] propose a technique for detecting operator errors by performing upgrades or configuration changes in a "validation slice," isolated from the production system. The upgraded components are tested using the live workload or pre-recorded traces. This approach requires component-specific inbound- and outbound-proxies for recording and replaying the requests and replies received by each component-under-upgrade. If changes span more than one node, multiple components (excluding the database) can be validated at the same time. Oliveira *et al.* [13] extend this approach by performing change operations on an up-to-date replica of the production database. Because these approaches operate at component granularity, they require knowledge of the system's architecture and queuing paths, and some errors remain latent if the components are tested in isolation [12]. Moreover, implementing the inbound- and outbound-proxies requires an understanding of each component's behavior, *e.g.*, the communication protocols used and its non-determinism. For instance, routing requests to a different application server in the validation slice would produce equivalent results, but processing database transactions in a different order would compromise the replication. To enforce a common order of execution, database requests must be serialized in order to prevent transaction concurrency, for both the production database and the validation slice [13]. Aside from inducing a performance penalty during the upgrade, this intrusive technique prevents testing the upgrade's impact on the concurrency-control mechanisms of the database, which limits the usefulness of the validation results. Compared with these approaches, Imago does not change the way requests are processed in the production system and only requires knowledge of the ingress and egress points.

The other components of the system-under-upgrade and the internal queuing paths are treated as a black box. Unlike the previous approaches, Imago targets end-to-end upgrades of distributed systems, and it addresses the problem of coordinating the switchover to the new version. Moreover, Imago's design facilitates upgrades that require long-running, computationally-intensive conversions to a new data format.

6.4 Dependability Benchmarking for Upgrade Mechanisms

Evaluations of most of the previous upgrade mechanisms focus on the types of changes supported and on the overhead imposed, rather than on the upgrade dependability. Because of this reason, while the costs of upgrading techniques (*e.g.*, atomic upgrades, isolation between the old and new versions) can be assessed in a straightforward manner, their benefits are not well understood. User studies [12], fault injection [12, 13] and simulation [1] have been used to assess the effectiveness of previous approaches in reducing the number of upgrade failures. We rely on our upgrade-centric fault model to perform systematic fault-injection experiments, with an improved coverage of upgrade faults. We inject faults manually, in order to determine the impact of each fault type on the three upgrading approaches compared, and we also use an existing fault-injection tool for automatically injecting Type 1 faults. Similar fault-injection tools can be developed for upgrade faults of Types 2–4, in order to evaluate the dependability of future upgrade mechanisms.

7 Conclusions

We propose a new fault model for upgrades in enterprise systems, with four types of faults. The impact of Type 3 faults (broken environmental dependencies) seems to be easy to detect using existing techniques. Faults of Type 1, 2, and 4 frequently break hidden dependencies in the system-under-upgrade. Existing mechanisms for online upgrade are vulnerable to these faults because even localized failures might have a global impact on the system. We present the design and implementation of Imago, a system for upgrading three-tier, enterprise systems online, despite hidden dependencies. Imago performs the end-to-end upgrade as an atomic operation and does not rely on dependency-tracking, but it requires additional hardware and storage space. The upgrade duration is comparable to that of an offline upgrade, and Imago can switch over to the new version without data loss and, during off-peak windows, without disallowing any client requests. Manual and automated fault-injection experiments suggest that Imago improves the dependability of the system-under-upgrade by eliminating the single points of failure for upgrade faults.

Acknowledgments. We thank Dan Siewiorek, Greg Ganger, Bruce Maggs, and Asit Dan for their feedback during the early stages of this research. We also thank Lorenzo Keller for providing assistance with the use of ConfErr.

References

1. Crameri, O., et al.: Staged deployment in Mirage, an integrated software upgrade testing and distribution system. In: Symposium on Operating Systems Principles, Stevenson, WA (Oct 2007) 221–236
2. Neumann, P., et al.: America Offline. The Risks Digest **18**(30–31) (Aug 8–9 1996) `http://catless.ncl.ac.uk/Risks/18.30.html`.
3. Koch, C.: AT&T Wireless self-destructs. CIO Magazine (Apr 2004) `http://www.cio.com/archive/041504/wireless.html`.
4. Wears, R.L., Cook, R.I., Perry, S.J.: Automation, interaction, complexity, and failure : A case study. Reliability Engineering and System Safety **91**(12) (Dec 2006) 1494–1501
5. Di Cosmo, R.: Report on formal management of software dependencies. Technical report, INRIA (Sep 2005) (EDOS Project Deliverable WP2-D2.1).
6. Office of Government Commerce: Service Transition. Information Technology Infrastructure Library (ITIL). (2007)

7. Oracle Corporation: Database rolling upgrade using Data Guard SQL Apply. Maximum Availability Architecture White Paper (Dec 2008)
8. : Oxford English Dictionary. 2nd edn. Oxford University Press (1989) http://www.oed.com.
9. Brewer, E.A.: Lessons from giant-scale services. IEEE Internet Computing **5**(4) (2001) 46–55
10. Oppenheimer, D., Ganapathi, A., Patterson, D.A.: Why do Internet services fail, and what can be done about it? In: USENIX Symposium on Internet Technologies and Systems, Seattle, WA (Mar 2003)
11. Keller, L., Upadhyaya, P., Candea, G.: ConfErr: A tool for assessing resilience to human configuration errors. In: International Conference on Dependable Systems and Networks, Anchorage, AK (Jun 2008)
12. Nagaraja, K., et al.: Understanding and dealing with operator mistakes in Internet services. In: USENIX Symposium on Operating Systems Design and Implementation, San Francisco, CA (Dec 2004) 61–76
13. Oliveira, F., et al.: Understanding and validating database system administration. USENIX Annual Technical Conference (Jun 2006)
14. Dumitraş, T., Kavulya, S., Narasimhan, P.: A fault model for upgrades in distributed systems. Technical Report CMU-PDL-08-115, Carnegie Mellon University (2008)
15. Kaufman, L., Rousseeuw, P.J.: Finding Groups in Data: an Introduction to Cluster Analysis. Wiley Series in Probability and Mathematical Statistics. Wiley (1990)
16. Sullivan, M., Chillarege, R.: Software defects and their impact on system availability-a study of field failures in operating systems. In: Fault-Tolerant Computing Symposium. (1991) 2–9
17. Chatfield, C.: Statistics for Technology: A Course in Applied Statistics. 3rd edn. Chapman & Hall/CRC (1983)
18. Dig, D., Comertoglu, C., Marinov, D., Johnson, R.: Automated detection of refactorings in evolving components. In: European Conference on Object-Oriented Programming, Nantes, France (Jul 2006) 404–428
19. Anderson, R.: The end of DLL Hell. MSDN Magazine (Jan 2000)
20. Di Cosmo, R., Zacchiroli, S., Trezentos, P.: Package upgrades in FOSS distributions: details and challenges. In: Workshop on Hot Topics in Software Upgrades. (Oct 2008)
21. Menascé, D.: TPC-W: A benchmark for e-commerce. IEEE Internet Computing **6**(3) (May/Jun 2002) 83–87
22. Dumitraş, T., Tan, J., Gho, Z., Narasimhan, P.: No more HotDependencies: Toward dependency-agnostic upgrades in distributed systems. In: Workshop on Hot Topics in System Dependability, Edinburgh, Scotland (Jun 2007)
23. Amir, Y., Danilov, C., Stanton, J.: A low latency, loss tolerant architecture and protocol for wide area group communication. In: International Conference on Dependable Systems and Networks, New York, NY (June 2000) 327–336
24. Amza, C., et al.: Specification and implementation of dynamic web site benchmarks. In: IEEE Workshop on Workload Characterization, Austin, TX (Nov 2002) 3–13 http://rubis.objectweb.org/.
25. Downing, A., Oracle Corporation. Personal communication (2008)
26. Boyapati, C., et al.: Lazy modular upgrades in persistent object stores. In: Object-Oriented Programing, Systems, Languages and Applications, Anaheim, CA (Oct 2003) 403–417
27. Zolti, I., Accenture. Personal communication (2006)
28. Neamtiu, I., Hicks, M., Stoyle, G., Oriol, M.: Practical dynamic software updating for C. In: ACM Conference on Programming Language Design and Implementation, Ottawa, Canada (Jun 2006) 72–83
29. Neamtiu, I., Hicks, M.: Safe and timely dynamic updates for multi-threaded programs. In: ACM Conference on Programming Language Design and Implementation, Dublin, Ireland (Jun 2009)
30. Lowell, D., Saito, Y., Samberg, E.: Devirtualizable virtual machines enabling general, single-node, online maintenance. In: International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, MA (Oct 2004) 211–223
31. Potter, S., Nieh, J.: Reducing downtime due to system maintenance and upgrades. In: Large Installation System Administration Conference, San Diego, CA (Dec 2005) 47–62