Partitioning Applications for Hybrid and Federated Clouds

Michael Smit, Mark Shtern, Bradley Simmons, Marin Litoiu*

York University
Toronto, ON, Canada

Abstract

On-demand access to computing resources as-a-service has the potential to allow enterprises to temporarily scale out of their private data center into the infrastructure of a public cloud provider during times of peak demand. However, concerns about privacy and security may limit the adoption of this technique. We describe an approach to partitioning a software application (particularly a client-facing web application) into components that can be run in the public cloud and components that should remain in the private data center. Static code analysis is used to automatically establish a partitioning based on low-effort input from the developer. Public and private versions of the application are created and deployed; at runtime, user navigation proceeds seamlessly with requests routed to the public or private data center as appropriate. We present implementations for both Java and PHP web applications, tested on sample applications.

1 Introduction

Enterprises are adopting cloud computing – particularly as a mechanism for organizing and accessing computing resources using a utility-based pricing model – to attain perceived benefits such as increased ability to scale, lower total cost of ownership (TCO), and the potential to improve availability. These resources are acquired on demand from a third-party (the *public cloud*), or can be managed within an organization in private data centers (the *private cloud*). A *hybrid cloud* uses public cloud resources to extend the capacity of a private cloud. Open-source and commercial software to enable private and hybrid clouds are increasing in popularity, and there is corporate interest in migrating workloads from the private cloud to a public cloud. A *federated cloud* includes multiple cloud providers or even tiered cloud providers; workloads may be moved among providers or tiers in response to changing workloads, updated pricing, locality considerations, or other factors.

One of the advantages of on-demand resources, as offered by public, hybrid, and federated clouds, is the ability to scale (e.g. [9]) in response to changing workloads. If a workload increases – for example during a seasonal shopping period – resources can be acquired temporarily from a large pool of resources (sometimes called *cloud bursting*). Since a private cloud (or other arrangement of private IT assets) is necessarily limited in size by existing hardware and datacenters, augmenting private resources with public on-demand resources increases the elasticity and flexibility of enterprise applications. However, deploying to public computing resources has security and privacy implications, particularly for customer-facing web applications or applications intended to run only on internal networks. Sharing hardware resources among multiple tenants using virtualization is inherently difficult to secure [10, 20]. One type of approach employs hardware-driven trust mechanisms (e.g. [11]), though this requires provider cooperation. Another approach mitigates some security risks by employing mechanisms to ex-

Copyright © 2012 Michael Smit, Mark Shtern, Bradley Simmons, Marin Litoiu Permission to copy is hereby granted provided the original copyright notice is reproduced in copies made.

* This author is a Visiting Scientist with IBM Canada CAS Research, Markham, Ontario, Canada.

tend the private cloud to resources on the existing public cloud (e.g. [18]), but cannot guarantee total protection when the hardware is controlled by another organization.

We specify a new class of application, a *partitioned application*, which is a transformation of an existing application. The transformation creates multiple *portions*, grouping together units of code based on their commonality in one or more dimensions. This partitioning enables more precise management policies when moving applications (and their workloads) among various cloud providers and tiers. For example, consider hybrid public-private clouds. Such applications would be logically partitioned into portions that can execute on public resources and portions that should not leave the private data center (perhaps because the component handles private information or because the code itself is proprietary).

In this paper, we contribute a novel methodology for producing such applications by automatically partitioning existing applications based on both low-effort input from developers and static code analysis. The development of the application is not affected, and no manual migration effort is required. We describe mechanisms for deploying two different portions that together represent the original application to the hybrid public-private cloud, based on existing infrastructure and standards-compliant extensions. The two portions do not communicate directly; they access a shared data store and session manager to maintain unified state, but otherwise operate independently. We present an approach to routing requests at runtime that ensures requests are handled by the private cloud when necessary (Section 2).

To verify the potential of partitioned applications and our methodology, we implement application partitioning tools for web applications and use them to partition and deploy both a Java EE application and a PHP web-based application (Section 3).

More generally, a key challenge when moving workloads and applications among multiple heterogeneous clouds is the differences among them. The components of an application might have (for example) different privacy/security demands, IO needs, CPU requirements, or latency expectations. Different cloud computing services provide different privacy guarantees, IO performance, CPU power, or network latency. Partitioning applications using our methodology will allow fine-grained deployment decisions at the application component level that can allow applications to make use of different cloud advantages without re-architecting or migration effort. Our partitioning approach allows us to consider a componentization of the application along the dimensions we are concerned about (privacy, security, IO performance, etc.) instead of the logical functionality groupings normally used. We discuss the potential of this class of applications further in Section 4.

Finally, we discuss related work in Section 5, and conclude the paper in Section 6.

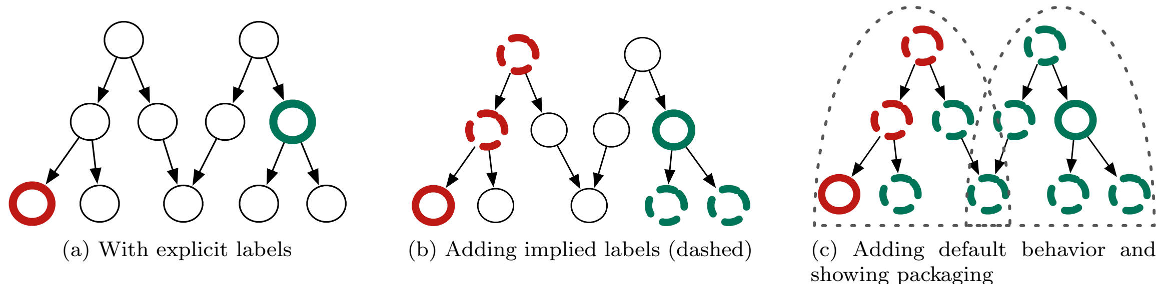
2 Methodology

Our novel process for running partitioned applications on a hybrid or federated cloud involves two primary stages: partitioning, where the application is partitioned into two or more portions; and deployment, where the portions are deployed to separate locations but linked together securely to provide shared functionality, with incoming requests routed to the correct deployment location. These stages are not entirely de-coupled; decisions made at either stage will have ramifications for the other. We'll describe the partitioning in the context of a private/public partitioning for a private/public cloud, as this offers a binary partitioning for a clear use case. The methodology – and implementations – apply to a variety of partitionings and use case scenarios.

2.1 Partitioning

We consider an application to be a collection of logical *code units*. The exact definition of a code unit is determined by application developers, the development language(s), and/or the particular implementation of this methodology. It should be a natural decomposition of the application, for example classes, methods/functions, files, fields, global variables/constants, etc. The partitioning problem[1] is determining which code units are re-

[1] This is not a strict partitioning, as there is overlap between the two portions.



(a) With explicit labels

(b) Adding implied labels (dashed)

(c) Adding default behavior and showing packaging

Figure 1: A simple dependency graph, with labels (mobile in green, immobile in red) from various steps of the methodology.

quired for the private portion, and which are required for the public portion (some code units may be present in both partitions for dependency reasons). The solution involves four steps: annotation, dependency detection, cascading labelling, and application transformation.

The application developer is asked to **annotate** the code units to provide cues to the partitioning algorithm. They need only annotate the code about which they have strong feelings; for example, if the only concern was that due to the PCI (Payment Card Industry) security standards some credit-card handling code cannot run in the public cloud [21], those code units would be annotated and nothing else. We define a set of five available annotations, two to provide cues on public cloud versus private cloud, and three optional cues on the structure of the application:

Immobile[2]: Code units that *must not* be moved out of the private data center.

Mobile: Code units that *may* be moved out of the private data center when needed.

DependsOn(<item>): Expresses that the annotated code unit depends on <item>. The <item> may be another code unit, or the keyword mobile or immobile. While most dependencies on other code items will be detected automatically in subsequent steps, this allows the developer to explicitly define dependencies using the higher-level constructs of mobile or immobile code units in general.

DependedOnBy(<item>): Expresses that <item> depends on the annotated code unit. In addition to the possible values for the DependsOn annotation, the keyword state may be specified to indicate that this code unit is used in the stored state of the application or session of the user.

Front Controller: Used to identify code units that are part of the implementation of a front controller design pattern [3]. This design pattern is popular in web applications, and uses a single url (with associated code) to handle all incoming requests. The receiving code unit is responsible for interpreting the request, managing session, establishing global properties (like database connections), and providing this information when executing the appropriate code units for the incoming request. This pattern is built into Java EE servers (and the standard), but is often implemented at the application level in other web programming languages. This annotation is a suggestion to ignore this code when analyzing dependencies, as while statically it depends on all other code units, at run-time it will only depend on code units required to respond to the given request.

The annotations can be specified using Annotations[3] when there is language support for this mechanism (in §3 we describe a set of Java

[2] While the annotations could be described in terms of public / private, those keywords are already used extensively in many programming languages; for clarity, we will use mobile/immobile to identify code that (respectively) may and must not leave the private data center. The use of terms that describe whether code can be moved or not also highlights the generality of this approach: while the illustrative motivation for making code immobile is security/privacy, there can be many other motivations.

[3] Note the difference between the general (lowercase a) and specific Java (uppercase A) annotations

Annotations we have implemented for Java developers). Conceptually, Annotations allow developers to annotate from within the source code, which makes the annotations explicit when developing or maintaining the source. The annotations can also be specified using configuration files, which does not require language support but is less visible during maintenance tasks.

In the **dependency detection** step, we use static code analysis [4] to identify the dependencies among the code units. Each code unit has inbound dependencies (code units that depend on it), and outbound dependencies (code units that it depends on). The result is a directed graph, where nodes are code units and directed edges indicate dependencies. Standard tools can be used for this step, though the output of standard tools must be augmented using the information expressed in the annotations: adding the dependencies specified using annotations, and removing dependencies relating to the front controller code units. Once the final directed graph is created, only the Mobile and Immobile annotations are relevant; the result of this step is a directed graph, with some nodes explicitly labelled Mobile and Immobile (Fig. 1a).

The **cascading labelling** algorithm labels the remaining nodes automatically based on the explicitly labelled nodes. In particular, any code units that depend on immobile code units are labelled ImpliedImmobile; these nodes cannot be moved to the public data center, as they rely on code that will not be there. Code units that mobile code units depend on are labelled ImpliedMobile, as a code unit isn't really mobile unless we can also move its dependencies. The same cascading approach is iteratively applied to the dependencies of the new implied mobile and implied immobile nodes labelled in the previous iteration. If the cascading labels conflict, the immobile labels take precedence and a warning is issued to notify the developer that some code units annotated mobile will not be mobile. Through this process, the labels propagate through the graph (Fig. 1b).

The result is a directed graph with five types of nodes: immobile and implied immobile, mobile and implied mobile, and unlabelled. The remaining unlabelled nodes are handled as per a default set by the developer: if the developer wishes to explicitly mark some code units as immobile and allow the rest to move, the default is mobile; if the developer wants most code to remain private, the default is immobile. Fig. 1c shows the application of a mobile default to the unlabelled nodes. The cascading labels and configurable defaults allow an application to be labelled with low developer effort.

Finally, **application transformation** applies the results of the graph labelling algorithm to the application itself, applying one transformation to produce the mobile portion, and a second to produce the immobile portion. The mobile portion must include only mobile and implied mobile code units, though it is not required to include all of them (if mobile code units are only called by immobile code units, they would be dead code in the mobile portion). To support request routing (describe in the next section, §2.2), the immobile code units are replaced by code units that respond to requests with a special response indicating there was an attempt to execute immobile code on the public cloud.

The immobile portion is more flexible; it can include only the code units marked immobile and their dependencies (which may be mobile or immobile), or it can include the entire code base unmodified, or it can include the entire code base with modifications to the immobile code units to set flags indicating the immobile code units have executed. (This latter option is helpful in the request routing algorithm.) One possible packaging is shown in Fig. 1c.

The exact approach to application transformation is somewhat language-dependent; for example, features such as late-binding, dynamic dispatch, reflection, and many others influence the options available to and the requirements for the transformation. We describe transformation methodologies for Java (§3.1) and PHP (§3.2) to illustrate some of the decisions.

2.2 Deployment

The deployment stage is responsible for taking two overlapping portions of the same application and deploying them to two separate infrastructures, linked only by a secure VPN

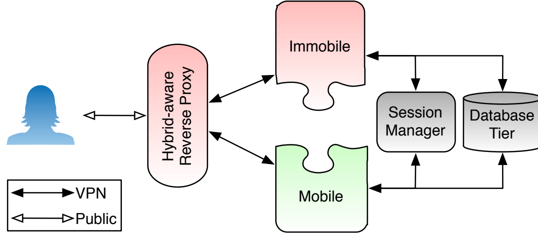


Figure 2: Abstract overview of deployment; private data center in red, public in green, gray components can be in either or both.

link[4], while presenting a seamless experience to the end user. The conceptual deployment architecture is shown in Fig. 2. A hybrid-aware reverse proxy is responsible for receiving and routing user requests (presenting a single unified front to the user), two portions of the same application (one on the public cloud and one in the private data center) receive requests, both portions can connect to a component responsible for managing and sharing sessions, and a database tier offers persistent storage. The session manager and database tier can be in either the private or public cloud, or both, depending on the application requirements.

Request routing requires inferring whether a request will require the execution of immobile code units. For some applications, a URL-based set of rules may be capable of routing requests correctly; requests for static content (css, images) are handled using this approach. However, there is not always a static mapping from a URL to the code units that are executed: internal redirects, dynamic dispatch, and front controller design patterns are just a few of the common elements of web applications that will invalidate this approach. Using static analysis, it may not be possible to accurately predict which code units will be exercised.

We describe an approach where we rely on (automatic) modifications to the public portion of the application to immediately flag and stop processing requests that attempt to execute immobile code units. These modifications are performed in the application transformation step of the partitioning stage.

As a general, language-independent approach, we describe a hybrid-aware reverse proxy (HARP). Without loss of generality, assume "mobile" is the default label for unlabelled code in the cascading labelling algorithm. By default, requests are routed to the public cloud. If the application attempts to execute immobile code units, instead our replacement code is executed. This code halts execution and returns an HTTP redirect to a specific, non-existent URL. If this redirect were sent to the client web browser, the application could break: client browsers do not re-send POST data when redirected, and some submitted information could be lost. In our case, the hybrid-aware reverse proxy detects this redirection and re-sends the request (including all POST, request, and cookie data) to the private cloud without involving the client at all. It appends a special cookie to the response, indicating that the session is now private; all future requests from that client will be directed to the private cloud.

A correct set of annotations will ensure that code units responsible for generating views that request private information are annotated immobile, which will ensure that a session is marked private before the user is even asked to enter private information. The deployer can decide to leave the session marked private for the remainder of the user's session, or can manually add code to signal the session can be moved back to the public cloud after a particular event (e.g. user logout). We can also automatically detect when the session can be safely transferred to the public cloud using the labelled dependency graph. We add a secondary label, which we call *Unencumbered*, which identifies any mobile code that has no interactions with immobile code (recall that immobile code can depend on mobile code). This unencumbered mobile code can be replaced with redirection code to the public cloud, updating the cookie appropriately.

An optimization to this approach allows us to redirect to the public cloud more aggressively, to reduce resource utilization on the private cloud. Some types of dependencies may not result in immobile code execution (for example, a child class could override all parent classes). By

[4]Optionally, the AERIE architecture [18] may be used to overlay a homogeneous environment with secure communication channels on top of the public cloud.

inserting special code into immobile code units in the immobile portion during the application transformation stage, we can ensure that each time a code unit is executed it sets a particular cookie value to false. On each request to the reverse proxy, the cookie value is incremented. If the cookie value reaches a certain threshold k, the HARP will understand this means no immobile code has executed in the last k requests, and it is safe to start routing requests to the public cloud. This decision can be made by the deployer per-application.

The hybrid-aware reverse proxy is still stateless: it acts as an intermediary for the client and maintains information about the request until a response is received from the application, but never has knowledge of more than that single request. Because the reverse proxy receives all information submitted by the user, it must run in the private cloud. It can perform load-balancing, and can be deployed in a tiered load balancer deployment.

Session management. As HTTP is a stateless protocol, web applications use sessions to manage the state of a client's interactions with the application. Typically each client is assigned a unique session ID (via a URL parameter, a hidden field in a web form, or a cookie); this session ID maps to session data stored on the application server (in memory, in files, in the database, or various other storage mechanisms). Session information may be stored client side, but this is typically not considered a best practice; if used, the information is encrypted.

When running a web application where requests are load-balanced to multiple servers, there are best-practices to ensuring requests are routed to a server that knows about the client's session. *Sticky sessions* is a practice where load balancers ensure that clients are always sent to the same server, which does not solve our problem. *Central session management* allows each application server to store session in a central repository rather than on the local server; for example, a central DBMS can be used for most applications, PHP allows a central memcached server[5], Oracle Coherence can store sessions centrally[6], etc. *Clustering* allows certain types of web application servers (Java EE in particular) to serve requests individually while storing some information globally. Each server implementation implements clustering differently; for example, in Apache Tomcat, sessions are multicast to all instances in the cluster.

Any of these approaches can be adapted to share sessions among private and public portions of an application. However, these solutions tend to focus on availability, so session information is not lost when a single instance goes down, and not on sharing sessions so requests can be handled by any server. They also assume multiple instances of an identical application are sharing sessions, not two separate portions of the same application. To help address this challenge, developers add an annotation to identify code units required for session management. We use this annotation to ensure that session management code units are available on both the public and private portions of the application.

Clusters assume the application is deployed once globally, and synchronized over multiple instances, so their use is not straightforward (see §3 for more details). With all approaches, care must be taken to manage synchronization of session information (for performance reasons, implementations default to synchronizing sessions to the global manager eventually, not necessarily before the request is returned to the client), session locking, and performance.

While our Java and PHP implementations use clustering and central session management, respectively, we believe future work should explore another option we call *just-in-time session transference*: when a web application looks up a session ID and does not find a result, a request is sent to the other portion requesting the given session information. The session data is transferred and request processing resumes. The advantage of this approach is it would allow run-time modification of session data to ensure that private data cannot "leak" from the private to the public cloud because the application stores sensitive information in the session data. The current approach does ensure that the session does not rely on immobile code units, though only if the optional

[5] `http://php.net/manual/en/memcached.sessions`

[6] `http://www.oracle.com/technetwork/middleware/coherence/overview/index.html`

`DependedOnBy(state)` annotation is used.

Database tier. To maintain consistency for web-based transactions, the application portions need access to the same data. How this is implemented is up to the deployer. The database tier can exist only in the private cloud, which is a functional approach but has potential performance issues (a local cache can be used to mitigate this problem). Full database replication to the public cloud could undermine the partitioning work at the application tier. A subset of the database could be replicated instead.

There is potential for further innovation in this area. For example, our dependency graph approach could be extended to include database-level dependencies (tables, columns, etc.) and only the required shards of the database could be replicated to the public cloud. Alternatively, data could be replicated to the public cloud only when it is first requested by the application running in the public cloud, relying on the correctness of the annotations to ensure that data requested by mobile code will not be private by definition. A NoSQL database could be extended to distribute information to the private or public data center based on some annotations.

For some deployments, **deploying the application in two separate portions** but with the same configuration and shared sessions/databases may be a technical challenge. For example, when using a Java EE cluster, a WAR file is deployed to a central controller that copies it to all running instances, which are assumed to be running the same application. Steps must be taken to modify the application post-deployment, then reload the instance without synchronizing with the rest of the cluster. In all cases, this process is highly language-dependent; we do not propose a general solution, but the following implementation sections will describe what was required to achieve this for Java EE and PHP applications.

3 Implementation

We implemented our methodology for two common web development platforms: Java EE, and PHP in a LAMP stack. We chose small but non-trivial applications to use as test cases for our implementation. The following sections describe the implementation of each of the stages and steps of our methodology, for each approach

3.1 Java Enterprise Edition

To illustrate the use of our Java implementation, we use a canonical example: the Duke's Bookstore from the Java EE tutorials[7]. It demonstrates the use of Java Server Faces (JSF), Java Server Pages (JSP), Java Servlets, various types of Java Beans, and database persistence via an e-commerce bookstore. It is approximately 2,500 lines of Java code, plus a set of JSPs, images, CSS files, etc.

3.1.1 Partitioning

Java applications are **annotated** using Java Annotations. We have implemented a library of Annotations based on the five types of annotations described in §2 that can be used to annotate Java source code. We offer Annotations at both the class level and the method level.

We annotated the bookstore application at the class-level, identifying one class that was important to storing session information (ShoppingCart)[8], one class that was responsible for managing the checkout process (CashierBean), and a class responsible for handling credit cards (CreditCardConverter). The desired behavior was to allow all activities to proceed in the public cloud, moving to the private cloud only on checkout.

To **detect dependencies**, we modified the Dependency Finder application[9] to offer a programmatic API instead of operating as a command-line tool. We added the ability to parse `web.xml` and `faces-config.xml` files to detect dependencies between JSPs and Java source code. The resulting data structure can be traversed and queried for dependencies at the class, method, and field levels; when annotating at only one of those levels, the others are filtered out.

[7] `http://docs.oracle.com/javaee/5/tutorial/doc/bnaey.html`

[8] Note that this is a superset of annotating it Mobile, as session information must be shared among the public and private cloud.

[9] `http://depfind.sourceforge.net/`, Jean Tessier

We implemented the **cascading labelling** algorithm in the form of a command-line application that gathers the annotations from the source code, reads the dependencies from the programmatic API, and produces a labelled dependency graph and a set of files required for the mobile and immobile application portions. Developer-configurable options include the default label (mobile/immobile), the base package names, the location of the source files and configuration files, etc. The labelled dependency graph is visualized and presented to the user (using DOT) to allow them to modify their annotations and re-run the labelling tool. This visualization could be integrated into more interactive static analysis tools.

When run on our test application, the algorithm identified an issue with the sample application: a class that was annotated as important to state was not Serializable. Because all of the approaches to sharing session information require transmission over the network, serialization is required. Identifying this issue early in the partitioning process allowed us to make this minor modification to the code preventing future problems. After we corrected this issue, the algorithm produced a labelled dependency graph. This graph is visualized by the implementation using Graphviz, and presented to the developer for review (a fragment is shown in Fig. 3). Two JSPs were identified as implied immobile; other than our two explicitly immobile classes, the remainder of the application was either mobile or implied mobile.

Application transformation requires two primary modifications for each portion. For the mobile version, the JSF configuration file is updated to redirect navigation from immobile JSPs to special redirection pages, and the immobile JSPs are replaced with redirection code. These redirections point to a specific but meaningless URL that is understood by the HARP (`http://private`), which is required to properly route incoming requests where the URL is not sufficient to identify the appropriate destination. Second, an Ant task is created that, when executed, will produce a WAR file that entirely excludes code marked immobile. For the immobile version, redirection code is added to code labelled both Mobile and Unencumbered, and the WAR file includes all code.

3.1.2 Deployment

We deployed the test application using the immobile/mobile WAR files, using Oracle Glassfish v3 as the Java EE server and Apache Derby as the database, all running on Amazon EC2 instances. A single public node (an Amazon m1.medium instance) and a single private node (an Amazon m1.small instance) were placed in different availability zones. We implemented the hybrid-aware reverse proxy, using Apache's `mod_rewrite` to pre-process requests. The public and private nodes do not communicate directly; each had a VPN connection to shared components (HARP, session manager, database). In addition to securing communication, the VPN once established avoids networking issues. HARP was responsible for **routing requests** to the correct virtual private address based on the parameters described in §2. We chose for this application to mark the session as private as soon as immobile code was executed, and to remain on the private server for the remainder of the session. The **database tier** was a single DB deployed in the private cloud, with queries passed over the VPN.

To make **session** data accessible to both portions, the public and private Glassfish servers were run in clustering mode. Each server was configured to share session information globally, multicasting session state to the other servers in the cluster before responding to the enduser[10]. We performed deployments with other approaches, including storing encrypted session information client-side and in a central session manager, and concluded that clustering was the best approach in terms of scaling, availability, ease of deployment, and performance. However, the methodology is not constrained to use this approach, and other approaches to session management could be used instead.

Deploying two similar-but-not-equal WAR files to a Java EE cluster is non-trivial. The deployer must deploy the mobile WAR to the cluster; this deployment will be automatically replicated to all instances in the cluster. For each private instance, the deployer must replace the contents of the deployed WAR entirely with the contents of the immobile WAR.

[10]Multicasting is not possible on most public clouds without the use of a VPN.

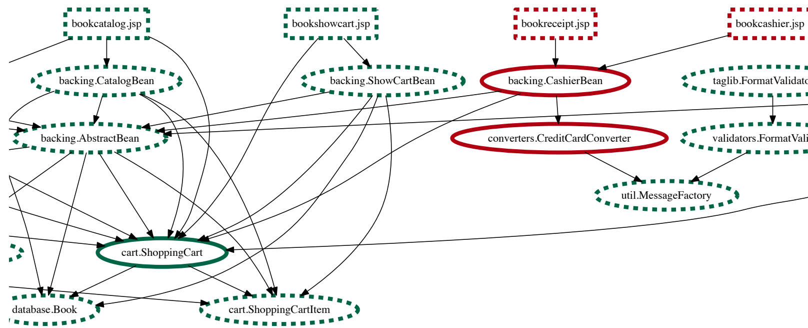


Figure 3: A fragment of the labelled dependency graph for the Java EE bookstore.

The instance must be restarted or reloaded, but prohibited from synchronizing with the rest of the cluster on startup (Glassfish, for instance, synchronizes the application configuration files with the original WAR file by default when an instance is restarted; this can be avoided using a command-line option).

3.1.3 Evaluation

To evaluate our implementation, we changed colours on the page of the private instance only, then executed a set of use cases manually. Recall that we had annotated credit card handling code as immobile. The use cases included browsing the e-store, viewing product details, signing in, viewing a shopping cart, editing a shopping cart, and checking out. Only the last use case involved credit card information. We noted at which points in the transaction we shifted to the immobile site, and verified that we were never prompted for credit card information while on the public server. We verified that the immobile code does not exist at the public instance; therefore, any request routing problems resulting in the attempted execution of immobile code would produce visible errors. No errors were encountered.

3.2 PHP

To illustrate the use of our PHP implementation, we used a framework for creating e-commerce applications in PHP called PHPShop, and the sample hardware store they provide. It uses a Front Controller design pattern and includes libraries for common tasks like shopping carts, session management, user management, and database connections. It totals approximately 10,000 lines of PHP code, plus associated HTML, CSS, Javascript, and image files. Where feasible, we made different design decisions from our implementation for Java applications to illustrate the alternative approaches to realizing our methodology.

3.2.1 Partitioning

PHP applications are **annotated** using separate standalone files (code-level annotations like Java Annotations are possible in PHP using third-party libraries; we chose this approach to demonstrate versatility). Annotations can be made at the function, method, class, or global level[11].

We annotated the hardware store at the function and class levels. The desired behavior was to allow all activities to proceed in the public cloud, moving to the private cloud only when credit card information would be handled. We identified four functions that handled credit card information, which we annotated immobile. We identified no classes being important to session information.

To **detect dependencies**, we used the phpCallGraph tool[12] to perform the static dependency analysis. The output was exported as JSON. The analysis examines dependencies among functions, methods, fields, and global constants. We modify the results to aggre-

[11] PHP allows code to exist in the global scope, outside of a class. To differentiate between functions defined globally and functions defined for a class, we will refer to the latter as methods (consistent with common practice).

[12] `http://phpcallgraph.sourceforge.net/`

gate methods and fields into the appropriate class. The tool cannot at this time examine code in the global scope that is not wrapped in a function. In our sample application, the global scope serves as a front controller, with application logic being encapsulated in classes and functions.

We used the same **cascading labelling** implementation used for Java applications. A data structure is created from the JSON which can be traversed and queried as required by the implementation, using the same data model as the Java implementation. As before, the result is a labelled dependency graph; from this, we produce a list of functions and classes that should be included in each of the private and public portions. A similar set of configuration options allow the user to set the default label for otherwise unlabelled code units.

When run on our test application, a labelled dependency graph was produced (Fig. 4). The nodes ending in parentheses are functions, the nodes in upper-case are global constants, and the remainder are classes. The graph shows that explicitly labelling the code that handles the sensitive information at the lowest levels identified 1 class and 3 methods as immobile.

Application transformation is built on the PHP Token Reflection library[13], which works on the source code level, in contrast to the internal PHP reflection library which works on interpreted code[14]. For the mobile portion, we locate the code units labelled immobile and replace the function contents (or the contents of all class methods) with custom redirection code pointing to the specific but meaningless URL which identifies processing should continue on the immobile site. For the immobile version, similar code replacement is done for code units labelled both Mobile and Unencumbered. As PHP is not pre-compiled, no further compilation is required.

3.2.2 Deployment

We deployed the test application using the immobile/mobile distributions, to two standard LAMP (Linux, Apache, Mysql, PHP) configurations on Amazon EC2. A single public node (an Amazon m1.medium instance) and a single private node (an Amazon m1.small instance) were placed in different availability zones. The two were connected via a VPN that limited network bandwidth to 10Mbps (to avoid any benefit due to being co-located).

We used the existing implementation of the hybrid-aware reverse proxy, in the same configuration. This application again was configured to mark the session as private as soon as immobile code was executed, and to remain on the private server for the remainder of the session.

PHP supports storing session data to an in-memory cache out-of-the-box. To make **session** data accessible to both portions, we use a distributed memcached service built on the same two nodes to which the application was deployed. The PHP interpreter checks this distributed cache for session data, which could come from any node. While this is an approach with language support and is used for large-scale PHP applications, other approaches to sharing session information are not precluded.

The **database tier** was a single MySQL database deployed in the private cloud, with queries passed over the VPN. **Deploying** two similar-but-not-equal PHP applications to multiple servers requires only that the correct archives be unpacked on the correct servers.

3.2.3 Evaluation

Evaluation proceeded as for the Java EE prototype. Visual modifications to the portion hosted on the private instance were used to assess when transactions had been forwarded to the private instance. We executed a set of standard e-commerce use cases (browse, add to cart, checkout, register) and verified that any time we submitted or viewed credit card information the pages were submitted to and handled by the private instance. We inspected the headers and cookies sent and set by both the HARP and the application to ensure requests were being routed properly. No errors were encountered.

4 Discussion

This paper has focused on the use of partitioned applications for hybrid private-public

[13] `https://github.com/Andrewsville/PHP-Token-Reflection`

[14] This tool is one of several that would allow big-A Annotations of PHP code.

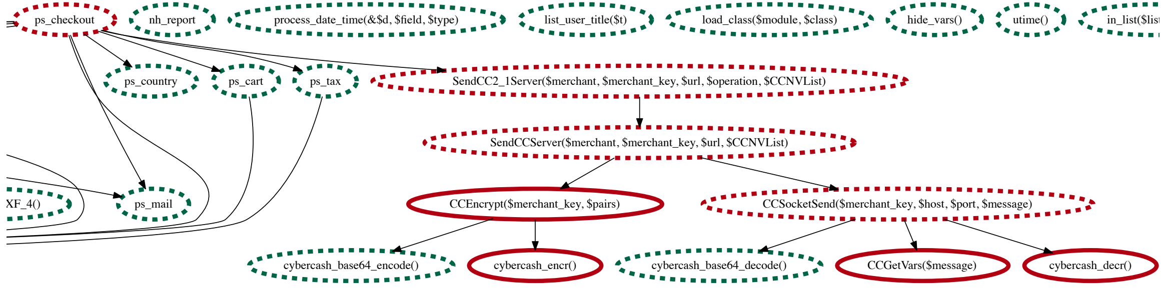


Figure 4: A fragment of the labelled dependency graph for the PHP hardware store.

clouds. Privacy/security is the *dimension* along which we partition the application. In this section, we will first discuss the challenges and limitations of our current approach. Second, we will discuss additional use cases and dimensions for partitioned applications, and third possible extensions and improvements to our methodology for creating partitioned applications.

4.1 Challenges and Limitations

When code is transformed after being written by the developer, there are implications for other aspects of the development process, like maintenance and debugging activities. The explicit annotations at the code level will assist developers in identifying code units that may change when partitioned, but further tool support may be required to support activities beyond development and deployment.

The stop-and-redirect approach implemented by HARP to help route requests has performance implications: the response to the users is delayed, and requests are made to two separate web servers. Though we have not quantified this delay, it could have implications on meeting service level agreements and perceived quality of service. Further analysis to quantify – and minimize – this impact is required.

4.2 Partitioned Applications

Computation-intensive operations. The use case of this paper resulted in a web application that was largely public, with a small number of components limited to a private location. A related use case would be a web application that is mostly private, except for some workload-intensive components. For example, consider an e-commerce application that purely runs data analytics to measure the effectiveness of advertisements, cross-selling, up-selling, A-B testing, and so forth. These tasks could be explicitly marked public and moved to the public cloud to reduce the impact on the limited resources in the private data center, while the remainder of the application runs on the private data center.

Tiered clouds. A national research project in Canada [17] proposes an architecture where virtualized resources exist close to end users (the *smart edge*), offering low-latency on-demand utility computing to any applications serving nearby users. Applications can move between the smart edge and traditional data centers (the *core*). Adaptive algorithms will manage this two-tier infrastructure, overseeing allocation of resources and the migration of applications automatically. An open question is determining which portions of an application can be moved to the edge, and which must remain in the core. The partitioning might be on the dimension of privacy and security (the ownership structure of the edge is not yet clear), or to which components require low-latency resources. We envision developer-driven annotations at development time that are used to deploy the application to the two-tiered edge-core cloud automatically at deployment time, or used to adaptively move portions of the application between the tiers.

Provider selection. Cloud brokers (e.g. [16]) and cloud abstraction layers[15] allows

[15] E.g. Deltacloud (`http://incubator.apache.org/`

run-time decisions regarding from which cloud provider to request resources. Applications could be partitioned along dimensions that provide cues for resource allocation. For instance, code could be annotated as running best on Intel hardware, or on providers that support temporary bursting to higher-than-allocated CPU levels. Or code that is truly stateless can be annotated as such, and deployed to spot instances (Amazon EC2 IaaS instances acquired at a bid price from otherwise idle resources and thus typically less expensive but also subject to unexpected termination).

4.3 Partitioning and Deploying

Dynamic analysis. The current cascading labelling algorithm uses static analysis in combination with developer cues to perform the partitioning. However, a substantial body of work exists regarding the dynamic analysis of software that could be leveraged to fine-tune the partitioning. For example, some workflows might have static dependencies on private code, but the private code might not be executed for several common workflows. Dynamic analysis of a deployed application could associate requests with workflows and make even more fine-grained decisions about application partitioning and request routing.

Data tagging. If the purpose of partitioning is to control where private data is handled, it may be possible to do tagging at the data level instead of at the code level. This could mean tagging and sensitivity ranking of flows of information through the application itself at runtime (*e.g.*, [19]), or could involve tagging code at the database abstraction level, for instance EJBs. Private data requested from a public location could result in query modification, data anonymization, or a redirect to a private location.

Loose coupling. The current approach processes requests entirely in one location or another: if a request does not invoke immobile code, it runs entirely in the private location. An extension would be to route all requests to the private location. Code marked mobile is deployed to the public location, but is refactored to be accessible via a web services interface.

Annotation at higher level of abstraction. While the cascading labelling algorithm can substantially reduce the annotation effort of the developer, for very large applications or when developers wish to explicitly annotate all code units, the workload can be further reduced by allowing annotations at a higher level of abstraction: for example, UML diagrams.

5 Related Work

The approach described in this paper partitions the application. When the motivation for partitioning is the privacy and security of information, another approach would be to partition the data and enforce the partitioning at the database layer. Application partitioning is more useful for other use cases (for example, when the code units are themselves private), but is a useful first step to data partitioning. For a transactional web application, the challenges of request routing must be resolved; the partitioning and deployment stages of this paper provide a resolution. We believe that ultimately the annotation step will happen at the database level (perhaps using a database with privacy data labels such as a Hippocratic database [2]). There is an existing body of work in partitioning databases (often for performance reasons [7]) that can be leveraged, as well as work on tracking information flows through an application [15, 19, 22].

Providing slightly different versions of an application is related to *conditional compilation*, which can be implemented via a variety of mechanisms (for example, [1] discusses the use of both pre-processor directives like `#ifdef` and aspect-oriented programming). Conditional compilation produces different executables based on the conditions in which the code is compiled, for example including different low-level libraries when compiled for Windows versus for Linux. The developer writes each of the various implementations, and uses compile-time mechanisms to allow the compiler to choose which implementation to compile. Our approach requires one implementation which is automatically transformed as needed. Partitioning is also conceptually different, as at run-time conditionally compiled

`deltacloud/`), Libcloud (`http://libcloud.apache.org/`), Simple Cloud (`http://www.simplecloud.org/`), fog (`http://fog.io/`).

programs still execute as a unified whole.

The Wedge tool [5] splits an application into compartments with different privilege levels. By limiting the communication between compartments of different privileges, and separating components that deal with arbitrary user input from those executing core logic, several security threats can be mitigated. They provide tools for analyzing the call graph to identify what minimum level of privileges are required for each compartment, and primitives to implement the splitting. The implementation is still performed by the developer, using the tools to assist them.

Microsoft's Volta project [14] took code written in a .NET language using Visual Studio and transformed it into server-client distributed architecture, based on annotations added by the developer. Microsoft is no longer publicly supporting this tool.

Academic research into hybrid clouds has focused on the middleware / abstraction layers for creating, managing, and using hybrid clouds, though there are some projects looking at being aware of workload requirements. For example, Zhang et al. [23] used the MapReduce paradigm to split a data-intensive workload into mapping tasks sorted by the sensitivity of the data, with the most sensitive data being processed locally and the least sensitive processed in a public cloud.

Khadilkar et al. [12] describe a method for processing data retrieval queries on hybrid clouds, where subqueries are run on the public and private clouds separately with the results joined together to answer the initial query. This allows the query to be processed in a distributed fashion without compromising sensitive information.

Application partitioning is superficially similar to the problems of domain and function decomposition for parallel, distributed, or grid computing [6, 13]. The focus of decomposition is replacing sequential approaches with parallel implementations, rather than on splitting the responsibility of executing application logic to different components with differing levels of trust. Other approaches to splitting applications into components to run on untrusted platforms (e.g. grid computing) require redevelopment of the application in a specialized programming language (e.g., [8],[22]), rather than the modification of an existing application with low developer effort.

6 Conclusion

Increased flexibility when deploying to hybrid and federated clouds will be required as the cloud ecosystem continues to evolve. We propose *partitioned applications* as a low-effort approach to enabling deployment decisions at the level of code units, rather than at the application level. This enables applications to leverage on-demand public clouds where it would not otherwise be permissible.

We described a methodology for creating partitioned applications using static analysis with annotations from the developer. The approach is heavily automated and uses an algorithm called cascading labelling to minimize developer effort. An application can be packaged based on the labelled dependency graph produced during the partitioning step. We then presented the requirements of deploying a partitioned web application such as shared sessions and database access, and described several approaches to meeting these requirements. Our hybrid-aware reverse proxy (HARP) can automatically route incoming web requests to the correct portion of the application.

To demonstrate this methodology, we implemented it both for Java EE applications and PHP web applications. While key contributions like cascading labelling and the HARP were used for both without modification, we intentionally varied other aspects of our approach to demonstrate the flexibility of the methodology. A Java and a PHP e-commerce store were processed using the implementation; we showed that we could partition the applications such that code units that handled credit card information always and only executed in the private cloud, and that HARP routed requests between the private and public cloud seamlessly.

We believe this class of applications has potential for various compositions and federations of clouds, and for various use cases. Partitioned applications can be deployed and adapted using fine-grained management policies at the code-unit level, across various dimensions.

Acknowledgements

This research was supported by IBM Centres for Advanced Studies (CAS), the Natural Sciences and Engineering Research Council of Canada (NSERC) including through the SAVI project, Ontario Centre of Excellence (OCE) and Amazon Web Services (AWS).

About the Authors

Michael Smit is a postdoctoral fellow at York University in Toronto, Canada, and a member of the Adaptive Systems Research Lab led by Marin Litoiu. He completed his PhD with Eleni Stroulia at the University of Alberta in 2011. His research interests include adaptive computing, cloud technologies, software engineering, and service-oriented applications.

Mark Shtern completed his PhD with Vassilios Tzerpos at York University in Toronto, Canada. He is a postdoctoral fellow at York University in the Adaptive Systems Research Lab. His research interests include security, autonomic/adaptive system, cloud computing, reverse engineering, program comprehension, information retrieval and clustering.

Bradley Simmons is currently a Postdoctoral Fellow in the Adaptive Systems Research Group at York University in Toronto, Canada. He earned his B.Sc. in Biology with an Honors in Genetics, an M.Sc. in Computer Science and Ph.D. in Computer Science at the University of Western Ontario in London, Canada. His research interests include, but are not limited to, policy-based management of distributed systems, business driven IT management and the further development of the strategy-tree concept and its implementation.

Marin Litoiu is a professor at York University. Prior to that he was a Senior Research Staff Member with the Centre for Advanced Studies, IBM Toronto Lab, where he led the research programs in Autonomic Computing, System Management and Software Engineering. He was the Chair of the Board of CSER, a Canadian Consortium for Software Engineering Research and Director of Research for Centre of Excellence for Research in Adaptive Systems. Dr. Litoiu holds doctoral degrees from the University Polytechnic of Bucharest and from Carleton University. His research interests include autonomic computing; high performance software design; performance modeling, performance evaluation and capacity planning for distributed and real time systems.

References

[1] Bram Adams, Wolfgang De Meuter, Herman Tromp, and Ahmed E. Hassan. Can we refactor conditional compilation into aspects? In *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, AOSD '09, pages 243–254, New York, NY, USA, 2009.

[2] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Hippocratic databases. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 143–154, 2002.

[3] D. Alur, J. Crupi, and D. Malks. *Core J2EE patterns: best practices and design strategies*. Prentice Hall PTR, 2003.

[4] D. Binkley. Source code analysis: A road map. In *Future of Software Engineering, 2007. FOSE '07*, pages 104 –119, 2007.

[5] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: splitting applications into reduced-privilege compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 309–322, Berkeley, CA, USA, 2008.

[6] D. Callahan, K. Kennedy, et al. Automatic decomposition of scientific programs for parallel execution. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 63–76. ACM, 1987.

[7] S. Ceri, M. Negri, and G. Pelagatti. Horizontal data partitioning in database design. In *Proceedings of the ACM SIGMOD international conference on Management of data*, pages 128–136. ACM, 1982.

[8] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng,

and Xin Zheng. Secure web applications via automatic partitioning. *SIGOPS Oper. Syst. Rev.*, 41(6):31–44, 2007.

[9] H. Ghanbari, B. Simmons, M. Litoiu, and G. Iszlai. Exploring alternative approaches to implement an elasticity policy. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 716–723, 2011.

[10] W. A. Jansen. Cloud hooks: Security and privacy issues in cloud computing. In *Proc. 44th Hawaii Int System Sciences (HICSS) Conf*, pages 1–10, 2011.

[11] Seongwook Jin, Jeongseob Ahn, Sanghoon Cha, and Jaehyuk Huh. Architectural support for secure virtualization under a vulnerable hypervisor. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 272–283, New York, NY, USA, 2011.

[12] Vaibhav Khadilkar, Murat Kantarcioglu, Bhavani M. Thuraisingham, and Sharad Mehrotra. Secure data processing in a hybrid cloud. *CoRR*, abs/1105.1982, 2011.

[13] P. Lee and Z.M. Kedem. Automatic data and computation decomposition on distributed memory parallel computers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(1):1–50, 2002.

[14] D. Manolescu, B. Beckman, and B. Livshits. Volta: Developing distributed applications by recompiling. *Software, IEEE*, 25(5):53 –59, 2008.

[15] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9(4):410–442, 2000.

[16] Przemyslaw Pawluk, Bradley Simmons, Michael Smit, Marin Litoiu, and Serge Mankovski. Introducing STRATOS: A cloud broker service. In *2012 IEEE 5th International Conference on Cloud Computing (CLOUD)*, pages 891 –898, 2012.

[17] SAVI. Strategic network for smart applications on virtual infrastructure (savi). http://www.savinetwork.ca/, 2011. Last access 02/01/2012.

[18] Mark Shtern, Bradley Simmons, Michael Smit, and Marin Litoiu. An architecture for overlaying private clouds on public providers. In *8th International Conference on Network and Service Management, CNSM 2012, Las Vegas, USA*, 2012.

[19] Michael Smit, K. Lyons, M. McAllister, and J. Slonim. Detecting privacy infractions in applications: A framework and methodology. In *IEEE 6th International Conference on Mobile Adhoc and Sensor Systems, 2009.*, pages 694–701, 2009.

[20] Marten Van Dijk and Ari Juels. On the impossibility of cryptography alone for privacy-preserving cloud computing. In *Proceedings of the 5th USENIX conference on Hot topics in security*, pages 1–8, Berkeley, CA, USA, 2010.

[21] Virtualization Special Interest Group, PCI Security Standards Council. PCI data security standard (PCI DSS) - information supplement: PCI DSS virtualization guidelines. `https://www.pcisecuritystandards.org/documents/Virtualization_InfoSupp_v2.pdf`, June 2011.

[22] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Trans. Comput. Syst.*, 20(3):283–328, 2002.

[23] Kehuan Zhang, Xiaoyong Zhou, Yangyi Chen, XiaoFeng Wang, and Yaoping Ruan. Sedic: privacy-aware data intensive computing on hybrid clouds. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 515–526, New York, NY, USA, 2011.