# Top 9 rules for cloud applications

## The dos and don'ts of making your application cloud-ready

Kyle Brown
Mike Capern

April 09, 2014

An application is cloud-ready if you can effectively deploy it into a public or private cloud. That is, you must design the application so that it can leverage the platform-as-a-service (PaaS) layer on which it runs, and won't break because of design limitations that collide with assumptions that are made in the PaaS layer. If you follow these simple rules in your application design, you can usually make your existing applications cloud-ready without going through an entire reimplementation.

Preparing an application to run on the cloud is becoming a common task. How difficult a task that is varies widely depending upon how your application is written. A common distinction is between applications that are "cloud-ready" versus "cloud-centric" (sometimes called "born on the cloud").

Essentially, an application is **cloud-ready** if it can be effectively deployed into either a public or private cloud. That is, the application must be designed so that it can take advantage of the capabilities that are provided by the platform-as-a-service (PaaS) layer on which it runs. Likewise, the application should not break because of design limitations that collide with assumptions that are made in the PaaS layer.

For this reason, many developers push toward replacing traditional applications with entirely new applications that are built to be **cloud-centric**. These applications are often built by using different tools and runtimes than traditional applications. For example, if an application is being entirely redeveloped for the cloud, it might replace a relational database with a NoSQL database, like Cloudant or MongoDB.

However, you don't have to go so far as to abandon your entire existing tool and runtime suites. If you follow some simple rules in your application design, you can usually make your existing applications cloud-ready without having to go through an entire reimplementation. You can use these same rules as criteria for ranking your existing applications for migration to a dynamic cloud environment.

Here are nine rules for making your application cloud-ready.

## 1. Don't code your application directly to a specific topology

A key benefit of many cloud platform services is that they allow for immediate scalability changes in the application. This might be through true dynamic scalability, such as with the virtual applications in IBM® PureApplication® System, or by manually resizing the number of instances of an application – adding dynos in Heroku, or adding Warden containers in Cloud Foundry. The principle to remember is that if your topology can change, it will change. This is a radical shift! In a traditional environment, the application might assume a particular deployment topology (for example, a two-node IBM host names and host IP addresses. None of these assumptions are workable in a cloud application. Host names, IP addresses, and the number of application nodes in use can all change at a moment's notice. Assumptions about where "singletons" are in your topology can be especially problematic. If all the other nodes try to contact that particular node and it's not there—or even worse, if there are two of them—what happens to your application?

### What to do instead

The first rule must be to keep your application from being affected by dynamic scaling: Build your application to be as generic and stateless as possible. If you must use a singleton, enable a voting protocol so that the remaining nodes recreate a singleton if the singleton dies. Also, keep a permanent backup of the singleton's state in a shared repository, such as a database.

## 2. Don't assume the local file system is permanent

Because a node can be moved, taken away, or duplicated at any time, you can't make any assumptions about the longevity of files that are written to the file system. Suppose that an application uses the local file system as a cache of frequently accessed information. If the node is shut down and then restarted at a different location in a different VM, that cache will disappear, leading to different response times from different nodes in your topology.

### What to do instead

Instead of using the local file system as a store for temporary information, put temporary information in a remote store such as an SQL or NoSQL database. Be aware that reading static information from a file system is fine. For example, your application can read a configuration or properties file if each node has the same files in the same, or an equivalent, directory structure. Writing unique files to the file system gets you into trouble.

## 3. Don't keep session state in your application

Statefulness of any sort limits the scalability of an application—not just storing state on the local file system, but even storing permanent state in local memory. Unless the application can recover seamlessly from the removal of any node, and rebalance work instantaneously on the addition of a node, the application will have a hard time functioning in a cloud environment.

For many applications, the hardest type of state to eliminate is session state. It's so hard to eliminate that trying to do so entirely is often a fool's errand. It might be possible to store some state in the client browser in modern web applications (for example, by using the facilities

in HTML5). However, it's usually better to minimize the impact of that state by storing it in a centralized location, on the server. You must be careful in implementing that recommendation. In Java applications, HTTPSession state is often stored in-memory, which presents a problem if your entire application server can be added or removed at any time. Ruby on Rails uses a similar mechanism with its session[] hash, and the same issues apply.

### What to do instead

If you can't eliminate session state entirely, the best practice is to push it out to a highly available store that is external to your application server; that is, put it in a distributed caching store, such as IBM WebSphere Extreme Scale, Redis, or Memcached, or in an external database (a traditional SQL database or a NoSQL database).

## 4. Don't log to the file system

If you write your logs to the local file system, what happens in a crash that is so serious that it takes out the entire container or VM where your application was running? Or what if your PaaS layer decides to scale down your application and remove the VM or container entirely? In both cases, you lose valuable information for debugging problems, especially problems that began long before the user sees the first symptom.

### What to do instead

As a result of this issue, many PaaS layers, such as Heroku, Cloud Foundry, and PureApplication System, add log aggregators that can be redirected remotely. Or, you might prefer to use an open source aggregator, such as Scribe or Apache Flume, or a commercial product, such as Splunk. In any case, in a dynamic cloud environment, it's critical to have your logs available on a service that outlives the nodes that the logs were generated on. In this case, be aware of the destinations of your logs when you perform logging. Most log frameworks have different log levels that enable you to customize how much information is logged. If you know that your log information is going to be directed across the network, you might want to minimize the overhead of that traffic by reducing the log level to produce a manageable volume.

## 5. Don't assume any specific infrastructure dependency

This general principle has several manifestations. For example, you should not assume that the services that your application calls are at particular host names or IP addresses. Service-oriented architectures have been widely adopted in recent years, but it is still common to find applications that embed the details of the service endpoints they call. When those called peers or services can be relocated or regenerated within your cloud environment—and shift to new host names and IP addresses—the code of the calling applications breaks.

### What to do instead

Abstracting environment-specific dependencies into a set of property files is an improvement, but still inadequate. The problem with using files as a name is that you are constantly updating and changing properties files. Because applications need to be more resilient in a cloud environment, they should be agnostic to clustering. A better approach is to consult an external service registry

to resolve service endpoints, or delegate the entire routing function to a service bus or a load balancer with a virtual name.

## 6. Don't use infrastructure APIs from within your application

This is a rule with wide applicability because an "infrastructure API" can refer to a lot of different layers in your software stack. For example, many Java developers still create their own threads and manage their own thread pools, even though such concepts as the WorkManager API have been around for many years. The key advantage to avoiding a low-level infrastructural API is realized when the time comes to monitor your application. Existing monitoring tools will know about managed thread pools, but if you create your own, the cloud's monitoring tools are unable to aid you in discovering thread bottlenecks.

At the configuration level, ISVs are often inclined to make their applications as self-contained as possible. If they know that an application needs the TCP connection timeout at a low value, the app or its launcher script can verify or set this network option. A better practice now is for the application to delegate these requirements to the scripting that prepares the cloud environment for the application. That is, limit the range of APIs that are used in the application code. Also, shift responsibility for infrastructure services to the provider so that layers of the infrastructure—rather, the operating system image—can be updated without impact to the application.

In the management space, we've seen developers build application code that queries and manipulates the IBM WebSphere Application Server infrastructure through JMX APIs. This is great provided you precisely control your infrastructure. But, suppose that as part of your cloud migration, you move to a lightweight application server, such as the WebSphere Liberty profile. There, you'd have a different set of MBeans with different capabilities, which becomes another part of your code that must change. As we look to cloud APIs, such as OpenStack, the opportunity presents itself to try even more exotic options, such as building your own autoscaling through manipulating the OpenStack Nova APIs.

### What to do instead

This is possibly the most difficult of potential problems to remedy. When you start making assumptions about the infrastructure that your application runs on, it makes changing that infrastructure more challenging. Think about why your application code is calling an infrastructure service or API. Is this something that could move to the PaaS layer? In the JMX situation mentioned previously, the code queried the JMX APIs to provide a dashboard for application performance, which is something that a vendor solution, like ITCAM, could do more easily and portably. Now the question to ask is: Are there existing open source or commercial products that you can rely on instead? Your application should be concerned with solving the business problem that it's aimed at, and not with manipulating the infrastructure it runs on. Leave PaaS solutions in the PaaS layer, and keep them out of your application code.

## 7. Don't use obscure protocols

There are so many interesting protocols out there, and interesting packages built on top of them. The trouble is that they often take special configuration and tuning for resiliency. And, resiliency

is something that you really need in the cloud if you are going to add and remove nodes under a load. Why build in your own database connection model if the platform can provide it? Applications based on HTTP, SSL, and standard database, queuing, and web service connections are going to be more resilient in the long term, by delegating the configuration repertoire to the platform.

### What to do instead

If your application is using any older or non-standard protocols, now is the time to take this “silver lining” opportunity to modernize and standardize. For example, EJBs that used IIOP were cool at the turn of the millennium, but the world has moved on. Moving to an HTTP-based infrastructure based on such standards as REST (or even the older SOAP and WS-* standards) will make it easier to port your system to a new environment. It will also enable additional business opportunities that are provided by API management. Finally, you might want to consider that asynchronous protocols (such as IBM MQ or MQTT) are still alive and well and can be extremely effective for many styles of application programming. Rather than trying to make HTTP into something it isn't (like a reliable messaging system), take a minimalist approach, and apply the right tool for the job.

## 8. Don't rely on OS-specific features

It will come as no surprise that applications that use standards-based services and APIs are more portable to cloud environments than those that rely on specific operating system features. We often see a tendency to use OS-specific features when a higher-level, OS-neutral version is available. A simple example is for scheduling work to be done. Many application servers, such as WebSphere Application Server, build scheduling services directly into their APIs. Open source options, such as Quartz, are also readily available. However, many developers still invoke Java programs from OS-level schedulers like cron. This works fine if your application is running on Linux or another UNIX derivative. But, if you move to Microsoft® Windows®, you are out of luck. The principle works the other way, too. That is, why assume that the Windows Event Service is available to your application and preclude running in a Linux cloud?

### What to do instead

In some cases, you can remediate this by using compatibility libraries that make one operating system "look" like another. Cygwin is a good example of a compatibility library that provides a set of Linux tools in a Windows environment. Mono is a good example of a compatibility library that is going the other way to provide .NET capabilities in Linux. However, avoid the OS-specific dependencies as much as you can, and rely instead on services that are provided by your middleware infrastructure or your service providers.

## 9. Don't manually install your application

Cloud environments are quite likely to be created and destroyed more frequently than traditional environments. Your application will need to be installed frequently and on-demand. It follows that the installation process must be scripted and completely reliable, with configuration data externalized from the scripts. There are some ramifications with this. First, don't assume that a

user is present to accept a license agreement. Second, don't assume that a user will be available to choose between 1 of N different configuration options.

### What to do instead

At a minimum, capture your application installation as a set of operating-system-level scripts. If your middleware platform provides a built-in scripting mechanism (such as the Jython scripts that are available for WebSphere Application Server), take advantage of them. Keeping your application installation small and portable makes it easier for you to adapt to different automation techniques such as Chef, Puppet, or patterns in PureApplication System.

Ideally, also minimize the dependencies that are required by the application installation. For example, what is your minimum configuration? Does the database really need to be available to install the application? Or would a better option be for the application to start without its database, report the problem, and then increase function when the database becomes available?

## Conclusion

These simple rules will help you determine what it takes to get your applications ready for the cloud. If you're building an application that is "born on the cloud," take these rules to heart, and incorporate them directly into your application. If you're getting ready to move your application onto a cloud environment for the first time, taking the time to think about these rules and making the critical adjustments is a key first step along that road.

## Acknowledgements

Many thanks to Bobby Woolf for his patient review and editing of this article, and to James Kochuba for his helpful suggestions.

## Related topics

- Developing cloud-capable applications
- IBM Bluemix® fundamentals
- IBM developerWorks Cloud Computing: Articles, tools, and communities
- IBM developerWorks Middleware: Articles, tools, and communities