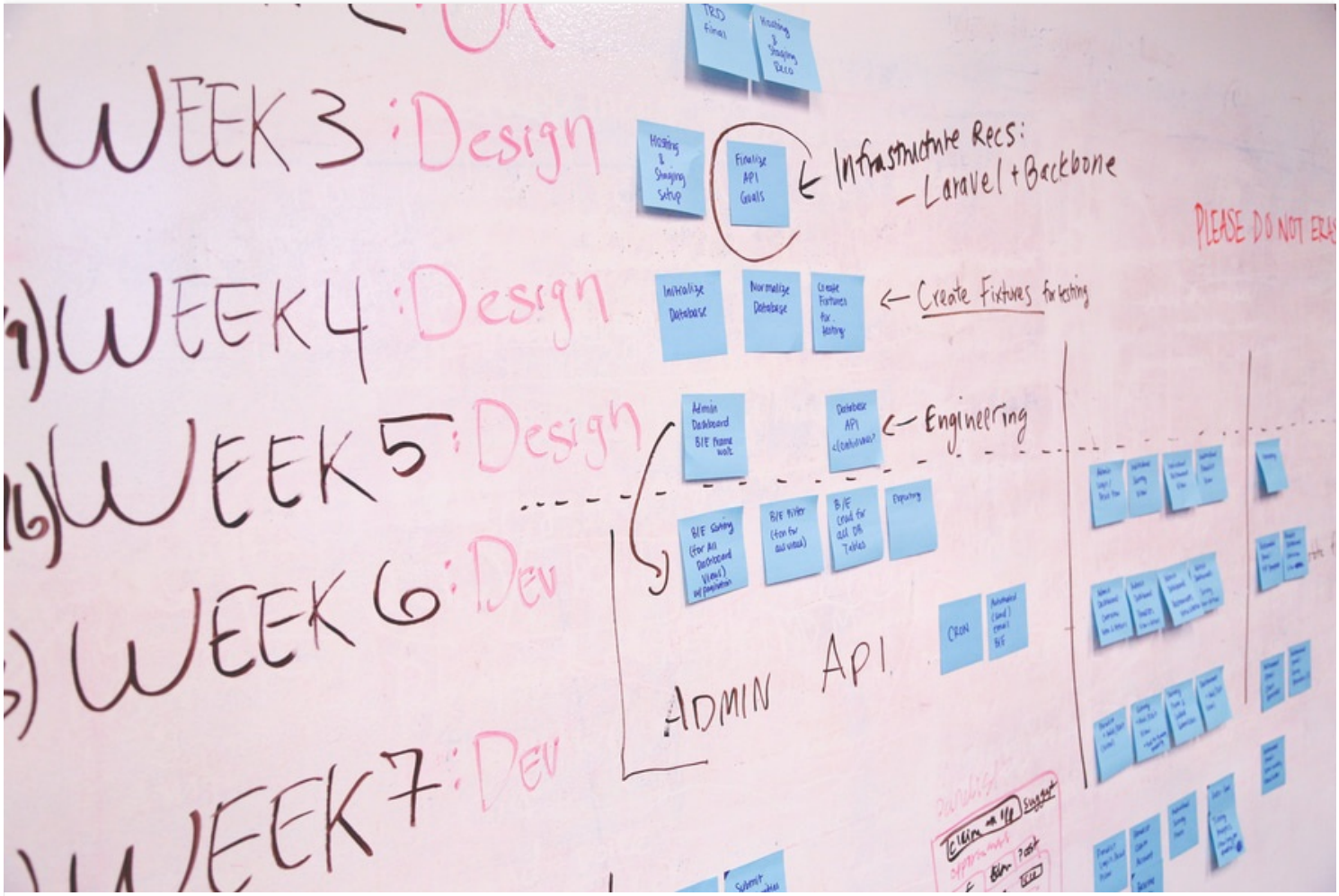

A DETAILED LOOK AT A BASIC CLOUD MIGRATION CHECKLIST

Patrick McClory May 24, 2016 Amazon Web Services, Automation, Business Strategy, Cloud Computing, DevOps
13,490 Views

I've written about design principles for Cloud before and while there's a lot of truth to the idea that applications work best when developed with the core design principles of the platform in mind – especially the concept of eventual consistency in AWS – the reality is that we all have a lot of applications that already exist.

Whether the applications are commercial, off-the-shelf, or custom deployments, there are a number of things to consider as you migrate to the public cloud – from performance to service-level capabilities and service-level agreements (SLAs). Rather than over-complicate things at the outset, let's start with a very basic cloud migration 'pre-flight checklist':

1. Identify areas where you can leverage higher-level cloud provider services like database, object storage, and other application-level managed services.

2. Understand and validate SLA requirements, Recovery Point Objective (RPO), and Recovery Time Objective (RTO) to ensure your needs line up with your cloud provider's capabilities.
3. Find your single points of failure and make sure you understand what the overall impact will be when it fails.
4. Have a plan to mitigate risk before and after your migration.
5. Know when and how to optimize cost.

Let's look at all of these in a more detail. First, while compute, network, and (block) storage are fairly ubiquitous elements of public cloud offerings, each provider offers their own set of more composed services. One very common offering is Database Management that includes automation around backup and recovery. These types of services can be novel, but they also have limitations.

One example: in the case of AWS' Relational Database Service (RDS) for SQL Server and Oracle, a number of non-core features aren't enabled when the instances and overall offering are managed on your behalf. When access to the SQL Server [master] table or the network-level utilities in Oracle are not available, this can become problematic for more advanced users. The time and effort required to reengineer the solution to remove dependencies far outweighs the benefits of a managed service and it can make your timeline for migration unacceptable.

While Amazon is great at continuing to listen to their customers and the list of unavailable features continues to dwindle, it's important to identify where you can leverage higher-level cloud provider services and where it's best to deploy that same software you have in your data center on compute instances.

Second, important to understand the full impact of failure and recovery in the cloud for your business. In the cloud there's no real way to hold your hardware vendor responsible when something fails, unlike in your data center or a physical machine. And even Amazon Chief Technology Officer and Vice President, Werner Vogels, is famous for being very upfront that 'everything fails, all the time'. One of the lessons Vogels says you must learn when deploying in the cloud is that responsibilities commonly placed at the hardware level are now moving 'up the stack' and should be mitigated more programmatically via automation.

Before making any decisions about your cloud migration, you need to identify your failure-threshold and ensure your cloud and managed service providers are able to meet your expectations. Understand your SLAs and what that means for your system's RTO and RPO goals. Keep in mind, meeting these expectations may require changing the architecture of your deployment, even with commercial software.

Microsoft Sharepoint is a great example of a solution that can be deployed in a fairly simple and streamlined manner. And while it's absolutely true that there are scale-up and scale-out deployments for Sharepoint that enable a significant amount of fault tolerance, that's not generally how I've seen it deployed in the real world. Meaning, to deploy Sharepoint to your cloud and also meet your specific SLA requirements, you may need a more complex deployment model and significant testing, all resulting in more money spent than you may have anticipated.

This brings me to the heart of the matter related to this discussion around failure – the third point on our pre-flight checklist. Single Points of Failure come in all shapes and sizes and have varying impacts to systems depending on how they're architected. Some are simple to find, such as a single network firewall device or a single instance that acts as a proxy to a third-party system. Others are more insidious, such as requiring sticky sessions on load balancers to handle application session data that's stored locally and in memory on your web or application servers. Resolving these kinds of concerns can be simple or crushingly difficult. Things like cluster election processes with a master node or well-known static IP's become potential hazards when failure and recovery of instances (automated or otherwise) cause significant thrashing of a system trying to coordinate distributed events or even finding a resource that's required for its runtime operations. Once you've found all of those single points of failure, the question is, just what do you do about it? This is where your SLA and RPO/RTO details come in. You need to be able to prioritize based on impact and then what the relative effort is to mitigate the concern.

In most cases, automation becomes key. A solid mix between pre-populated images and automation around the recovery of a system to it's last known good state is a very useful tool. At the same time, using old-school tools, like DNS, more dynamically starts to make a lot more sense for resources that have a tendency to move around. You'll want to make sure that components in your system respect things like time to live (TTL) and retry logic (Java is notorious for caching DNS queries indefinitely by default).

Configuration Management toolsets like Chef, Puppet and Ansible become a boon at this point because of their ability to dynamically index and inventory what's in your environment. These tools can also act on your cloud provider's Application Programming Interfaces (APIs) and Software Development Kits (SDKs) to orchestrate some pretty complex workflows that would otherwise take significant time and be subject to a lot of human error at a critical juncture (such as a 'system down' state). In addition, these tools blur the lines significantly between operational processes and software development capabilities. The newfound ability to test and validate these design choices well ahead of time on a system that isn't production (because, let's face it, no one likes testing in production) is an enormous game changer for most traditional operations organizations.

All of this allows you to make some very novel decisions as to what concerns you need to mitigate before migration and afterward – the fourth point in our checklist. The best advice I have here is to analyze your system's SLAs and assemble the foundational toolsets (configuration management, infrastructure as code, dynamic monitoring, etc.) to achieve a greater level of flexibility once you move to the cloud.

Finally, cost optimization is important. The big problem here is that premature optimization can lead to some surprisingly bad outcomes. What do I mean? If you're on a platform that links usage of a certain resource type to a pre-paid or lower-cost offering with a commitment over time (such as AWS' Reserved Instances or Azure Prepaid Subscriptions) and you choose the wrong size, you either A) spent too much or B) bought too little. If the latter, you'll either be bending over backwards to remediate that performance elsewhere or scaling up to achieve an acceptable level of performance.

A great example of this is in memory-intensive applications. What I've found is that, without proper load testing and some real-world burn-in time with the system running in production, aggressive choices related to instance sizing can mean that applications don't have enough memory to start or run properly. Test this well before you start and if you're deploying custom applications still under development, make sure that your test process accounts for this. Ensuring this is important so developers don't slip in new memory availability requirements without a thorough vetting on its' journey to production.

When it comes to cost optimization, 'measure twice and cut once' is the best way to approach the problem. Make sure you test the system in a way that is representative of the final production environment. This may mean that you pay full price for a period of time for the purpose of being able to best capitalize on the economics of the cloud over the long-term.

If you perform these reasonable sanity checks as you migrate into a public cloud setting, you are ultimately setting up your business for success. With a better understanding of your short and long-term goals, you are better prepared to migrate to and manage your new cloud environment. As Benjamin Franklin said, "by failing to prepare, you are preparing to fail."