# Getting to Elastic: Adapting a Legacy Vertical Application Environment for Scalability

*Eric Shamow* - Puppet Labs

## ABSTRACT

During my time in the field prior to joining Puppet Labs, I experienced several scenarios where I was asked to be prepared for so-called “elastic” operations, which would dynamically scale according to end-user demand. This demand only intensified as the notion of moving to IaaS became realistic. There's no button you hit marked "make elastic" to turn your infrastructure into an elastic cloud...rather you need to come to an understanding both of the technologies your organization uses, its tolerances for latency and downtime, as well as your platform, to get there. This paper discusses the key areas that must be addressed: organizational culture, technical policy development, and infrastructure readiness.

## Introduction

As I’ve moved through the industry, it’s become increasingly common to find organizations operating what might be termed an “internal cloud” - a commodity hardware infrastructure front-ended by VMware, Xen, or another virtualization technology, being used to cushion the need for rapid and varying server deployments. Over the past few years, I have seen increasing interest in outsourcing that operation - in moving to external cloud offerings including IaaS. In most cases, I've also needed to become prepared for elastic expansion of our apps as we modify them to scale out rather than up.

I encountered many of these problems during the time I spent as Manager of the Systems Operations group at Advance Internet. Advance is a mid-size company in the publishing field, running approximately 1050 servers in a local, private cloud. Although I left Advance prior to the full implementation of our elastic solution, I was deeply involved in the architecture and implementation of that solution, and was fortunate to learn valuable lessons about how to take an entrenched static environment into a dynamic one.

There's no button you hit marked "make elastic" to turn your infrastructure into an elastic cloud...rather you need to come to an understanding both of the technologies your organization uses, its tolerances for latency and downtime, as well as your platform, to get there. Advance traveled some of this road, and this report will include both information about the solutions we found, and some recommendations for those attempting to do the same.

## Characterizing the Problem

In order to consider what will be necessary to “go elastic,” we must first evaluate what that phrasing really means. How elastic do we want to be? What parts of our applications are able to scale easily? What parts do not? What elements of our process or infrastructure make automatic expansion impossible? In short, what do we need to know?

At Advance, in examining our environment, I identified five major questions or issues that would be show-stoppers for us implementing any kind of scalable environment:

1) Our servers and applications could not be deployed without human intervention. Documentation was limited and there was no automation available.
2) We had no information available about when to deploy a new server automatically. There was a mandate to be able to expand dynamically, but no information about what that meant.
3) Similarly, we did not know when to automatically retire a new server. How responsive to increases and decreases in load would we need to be?
4) What was to be the mechanism for the automatic deployment and retirement?
5) Were our applications optimized to take advantage of this type of scaling? In several cases our experience was that performance improvement was not a linear correlation with an increase in server count - and in fact that in some cases increasing parallelism was damaging to performance. We would need to determine which applications would need to be refactored to handle this architecture, and which were prepared to handle it natively.

In any environment facing similar issues, the five listed above will form the core of the matter - the remainder of our internal fact-finding extended naturally from the answers we found and the process we underwent in attempting to determine those answers.

For those undergoing the same exploration, this fact-finding exercise will form the groundwork for all future work in this space. This means that truthful responses and openness are absolutely necessary. The teams involved don't need to agree on a solution yet, but without a common understanding of the problem space, we cannot reasonably determine whose concerns or enthusiasm are justifiable. It can often help to present this as an opportunity to air long-unaddressed concerns in a new way. If the application team distrusts elasticity, encourage them to fully explain and justify those concerns and promise that they will be addressed as part of the proposed solution. Getting everyone to cooperate here is the most critical step of the process. For me, getting to elastic meant a lot less engineering than I expected, and a whole lot more PR, meetings, and assuaging of concerns.

## Elasticity Means Automation

The first key recognition about elastic expansion is that by definition, it means that the server provisioning process must be automated. This is a bridge that many organizations have yet to cross. In some cases the deployment process itself may be automated, but post-install configuration is not completed automatically. My own findings gathered from the organizations I have observed - and this was the case at Advance as much at others - are that most installation and configuration procedures are not automated because groups do not have clear and stable procedures that are followed for deployment. Whether this is because deployment teams do not maintain regular standards for system configuration, or because development teams do not provide accurate release notes or cleanly packaged applications ultimately comes down to finger-pointing; the organization as a whole must recognize that if it wants elasticity, it will need automation, and automation requires clarity of purpose and requirements, and stability of procedures.

Automation itself has multiple components, and depending on the breakdown of roles and responsibilities within an organization, these components are often managed by different groups. Infrastructure groups will have concerns about provisioning storage and network; OS groups will worry about package repositories, OS versioning, and configuration management; application groups will focus on updating application-specific configurations to recognize new or removed members of a cluster, reshuffling data that has been partitioned based on previous cluster size, and changing various application settings to properly tune performance. All of these are critical and should be clearly mapped.

Where possible, inquiry into how they affect each other is worth discussion - does repartitioning our data suggest different OS configs? With the new cluster size, should we alter our load balancer configuration? However, don't let these advanced discussions derail the primary goal of understanding how your systems are provisioned. The second-level analysis of how those systems interact will occur naturally during the design and implementation of your process, and should continue to iterate through its lifecycle. The most important thing is to come to an understanding of those manual processes which are not currently automated. Those manual steps are your hard roadblocks on the way to elasticity.

Ultimately, at Advance, we settled on a toolset of Kickstart for OS deployments, managed through Cobbler for the additional repository and profile information it permitted. We then handed off to Puppet for application installation and configuration, having worked closely with the application teams to build Puppet manifests that handled their applications appropriately. On the infrastructure side, the SAN, network and VMware team decided to manually script their deployment, resulting in a tool called vDeploy. I will discuss this tool later on in the paper. Ultimately, the tools you choose should be based on two factors: your own comfortability with them, and their flexibility to work well together and to integrate with each other. It is not always critical to choose the best-of-breed software, but rather to choose the software that best fits you and your organization.

## Elasticity Requires Open Metrics

An additional component to expanding and contracting an environment in an automated fashion is that accurate and relevant metrics about that environment must be available. In order for those metrics to be meaningful for elasticity, they must be reliable and comprehensive enough that an unattended system can make bottom-line decisions based on them: should I deploy or remove a live system from my customer-facing site immediately? This means that the metrics cannot be siloed as many IT reporting infrastructures are, but must reflect both the state of the application infrastructure as well as the applications running on it. These metrics must also be reliable: they must not be inaccurate, fudged, or intermittently available because of an individual group's desire to hide information from the rest of the team. Elastic expansions and contractions affect the whole without human intervention, but by definition this process is naive - it can only know what we tell it. If we lie to the system, the system will make poor choices.

The choice of metrics should also reflect a cross-disciplinary approach. Much is lost in IT monitoring because of a lack of communication between groups. A monitoring team will pride itself on implementing trend lines for disk utilization, but will fail to monitor a change in a transaction rate or size easily exposed by the monitored application itself. These metrics can predict an increase in the rate of growth at a time when the change would only appear to be a statistical anomaly in the storage data. Again, the discussions of these interrelationships will evolve from the discussions and implementations you are implementing here, and we shouldn't hesitate too long attempting to nail them down early. That said, any understanding we can get about interrelationships between the components in our environment helps us better predict future changes. Better prediction means better automation, which means elasticity that's less likely to break.

At Advance this was a major source of contention. Monitoring was highly siloed, with Systems controlling an array of Cacti, PNP, MRTG, and proprietary VMware, 3par, and NetApp applications to monitor and graph data - in fact, even within systems, monitoring was siloed, split between different implementations in the DBA, infrastructure, and operations spaces. Application development staff often maintained off-the-radar monitoring systems stashed on workstations or quasi-production servers. The metrics from these groups were never aggregated, and much time was lost bouncing requests and information back between multiple people who were hesitant to allow access to - or knowledge of the existence of - their proprietary systems.

## Openness Requires Culture Change

If the organization preparing to implement a model based on elastic expansion is not in the state needed to gather the information above - with a clear availability of infrastructure, OS, and application-level metrics across the board, honest communication between groups and well-documented deployment and configuration changes, elastic expansion is unlikely to be possible. These steps are all pre-requisites for technological change, but they themselves are less technological than cultural. If organizations are going to be prepared for elasticity - operating at a minimum cost most of the time but prepared for the huge onrush of traffic caused by an article "going viral" or the sudden success of their service[1], they must address the underlying lack of transparency before they can begin to work on the technical challenges.

In reality, getting this to happen is often the hardest part of the process. It is fortunate if the change is being implemented in a top-down manner, in that if management is mandating the change, it is often willing to enforce that mandate by requiring teams to cooperate. But what if the change isn't mandated?

In my own experience, the best approach is two-pronged. The first prong is to establish the missing communication. As the head of an Operations team, I regularly met with the head of Development teams, including those of small development groups that my predecessors had often ignored. I wanted to know their pain points, where Operations was letting them down or frustrating their work. Establishing this communication was key to establishing trust.

Trust, however, does not come through words but through deeds. The best action I found I could take in this regard was to surrender unilaterally. I might not be able to get developers or infrastructure to share everything with me, but I would share everything with them. Every incident was clearly documented, metrics were available to all teams, and we developed a process for requesting the addition of new metrics. I committed to making these newly-requested metrics available to them with an response time based on severity, reaching from 20-30 minutes during a crisis, to a maximum of 48 hours outside of one.

I also worked hard to develop a professional chain of command-based communication system with development managers. This may not be applicable in all engineering environments - in many having all

[1] http://blog.pinboard.in/2011/03/anatomy_of_a_crushing/

discussions on a public list is part of the fabric of their work culture. But it can also result in decisions made based on ego and pride rather than technical judgment. Being called out on an error or disagreement in public forces a different type of response from a concern brought quietly in private. At Advance I committed to bring development concerns to the relevant managers and help triage my team's issues rather than exposing them on our internal IRC channels and mailing lists, and asked the development managers to do the same. The ratcheting-down of public tensions combined with the daily give-and-take of triaging priorities with the other managers aided greatly in establishing an understanding of other teams' needs and willingness to cooperate.

## Getting Things Started

We've now established communication between departments, established some baseline metrics that we need to pay attention to, and defined clearly the expectation that server rollouts and retirements - from the bare metal phase to appearing in a user-facing cluster - should be automated. Now we're ready to do some work. But where to begin work?

For the purposes of this paper, I will assume that metric collection systems are already available to you, and that you need only tune your existing system to provide you the agreed-upon information. There are a variety of tools excellent at collecting and displaying raw data - from the simplicity of MRTG to more complex tools such as Munin or Cacti, and newer distributed tools such as Graphite or Ganglia. The use of one or more of these will depend on your data sources and the familiarity of your teams with the tools in question. My team used a mix of Cacti and PNP4Nagios, although we were strongly looking into Graphite as a replacement.

## Finding Meaningful Metrics

Assuming that we have monitoring technology in place, the next obvious question is "what do we measure?" The answer to this question may at first seem obvious to stakeholders on all sides of the discussion, but a quick synchronization of expectations often indicates that each group's answer is different. The infrastructure and OS groups will tend to monitor metrics focused on the performance of the system itself such as processor load, memory availability, I/O throughput, CPU percentage (distinct from load, which really measures queue length - a distinction lost on many involved in resource monitoring)[2], and swap usage.

In the meantime, the application team will likely be focusing on internal data points that reflect the actual capacity of the application itself, identifying performance of key areas of code, headroom left in caching applications such as Memcache or Varnish, and other data points that reflect how pieces of the code are relating to each other. If there is a separate business owner with access to a dashboard or metrics, that person or group is likely examining more vanilla performance stats - for a web application, time for first byte download, hits per second, and so forth.

It is very likely that none of these metrics will give you on its own the answer that indicates at what point your application will need to elastically expand. In fact, it is likely that, until this point, any discussions about non-elastic expansion have involved meetings between several stakeholders to review this data and find ways to optimize on existing hardware.

[2] http://blog.scoutapp.com/articles/2009/07/31/understanding-load-averages

Finding the right formula is an exercise in looking at aggregate data patterns, finding correlations that seem to reliably suggest the need for additional servers, and then regularly re-examining those metrics as the application and hardware profiles change.

The worst mistake you can make at this point is assuming that you know or understand too much about your application or environment. What was true several months ago may not be true now...a feature in the application that caused an I/O bottleneck six weeks ago may have been rectified in the application code four weeks ago, and now you've hit a CPU limit on your storage device. Assumptions about causes are more likely to cause bad interpretation of data, which in turn are more likely to cause a misunderstanding of what criteria will need to be used for automated scaling. So the important part of this stage is to have a fresh discussion about application performance and possible bottlenecks at all levels - an informed discussion, but one that makes no assumptions and thoroughly re-examines every facet of the environment looking for hidden indicators and bottlenecks. You won't find them all, but application, operations and business people together will find a lot more than any of those three alone. This is what DevOps looks like in practice.

## The Shifting Landscape

Before moving on to the next stage of enabling the elastic environment, I want to return to a phrase used just a few paragraphs back: "What was true several months ago may not be true now."

While this will always be the case is fast-moving, multifaceted IT environments, what should not be the case is that any of this changing truth should be undocumented, or worse a complete surprise to all but one or two people. In a field with rampant hyper-specialization with limited training budgets and one or two "experts" in a given technology per group, it is almost inevitable that sub-pockets of activity have developed which are at least partially invisible, even to members of that pocket's own team.

This type of change is absolutely toxic to elastic expansion. Since all of the painstaking research and rule development you are doing is based around a shared understanding of the environment, changes to that environment that are not automated make it impossible to deploy a single additional node without manual intervention. For that reason, change management must be implemented for an elastic environment to succeed.

This may sound like a leap, but if you examine the nature of elasticity, the reasoning becomes clear. Elasticity is essentially a set of rules wrapped around automation -- a set of conditions under which automated procedures should take place. Automation itself is really nothing more than a form of machine-parseable and actionable documentation - we are taking yesterday's run book or wiki doc and turning it into a YAML file, but in the end we are writing documentation about how a system should be configured, and then using an application to verify compliance with that document.

Note that I did not say that change control was needed - merely change management[3]. As long as the changes made are compatible with the rest of the operating environment and do not interfere with its operation, those changes can be submitted without review. Whether it is wise to do so is a different matter, but don't attempt to bite off more than you can chew here - the framework for change management can be

[3] http://www.technologyexecutivesclub.com/Articles/management/artChangeControl.php

expanded to include change control later on. For now, the important thing is that any change that would affect the ability to automatically rebuild a system is made part of the server and app deployment processes.

## Policy

Even when using clearly defined metrics to signal the need for expansion, there are additional factors to consider. First, we must consider the statistical anomaly. If you are running a website, you don't want to scale to a thousand machines because a web crawler hit your site and began to index, or because a user wrote a bad script to fetch your page every millisecond. Similarly, we must consider how long it takes for a new server to come up. Depending on the nature of your environment, this can be very tricky. If load increases sharply, you may need a new server in under a minute. Even with well-automated deployment, a large database server can take five or more minutes to power up and build. If this is not fast enough to save your application from falling over, we have missed the point of the elastic expansion.

The reverse is also true. If load drops to nothing because of an ISP failure, we do not want our production cloud to shrink to its minimum size. We also don't want to power down servers we think we may need again in a few seconds or minutes.

It is the rules around making these determinations that I refer to as "policy" - not a formal organizational policy, but rather an internal technical policy explaining when and how fast you expand, when and how fast you contract, what the artificial limits to both of the above operations should be, and how we work around the elements of those operations that don't fit our environment.

There is no formula that can be generated for this outside of an examination of your own application's behavior and the metrics you should now be gathering. As an example, however, I can discuss the type of solutions we had envisioned at Advance.

For the particular example of database servers, we looked at a combination of server load, database server queue length, and slow query information from the database server, and latency and queue information from the application side, to determine that a new database server was needed. However a new database server could take in excess of ten minutes to provision, far too long to resolve a sudden explosion of activity.

Advance's solution was to mandate that, depending on cluster size, one to two database servers would be provisioned and immediately powered down as par[4]t of our cluster at minimum size. Every time new database servers were automatically provisioned, 1-2 extra servers would be provisioned and immediately powered off. When the need came for new servers, we could begin provisioning additional servers but simultaneously power up the 1-2 idle servers, providing relief to the application within a minute, while additional resources came on line. We employed this strategy in reverse while shutting systems down, decommissioning them but always leaving 1-2 systems powered down but not destroyed.

We also decided to implement several caps on growth and decommissioning to hedge against the possibility of failures in our metrics and formulas. We only allowed growth to proceed at a limited rate, controlling the maximum number of servers that could be provisioned per 15-minute period, and setting a maximum limit on the number of machines that could be auto-deployed without administrator

[4] http://pulpproject.org/

intervention. We set similar limits on decommissioning.

This strategy works well for a “naive” application, where application servers are not aware of each other and can scale out horizontally. This is not the case for most applications, particularly in-house ones which have been written to scale vertically - requiring more resources such as RAM and CPU - rather than horizontally. As a result, many of these apps will not see a linear improvement as each server is added, and it is possible to see a diminishing return, and eventually even a negative impact from the addition of more servers. While an application rewrite down the line should help this, it’s almost never immediately possible; rather, you should tailor your expansion policies to fit the characteristics of the application you have, while encouraging your development teams to begin thinking in terms of horizontal rather than vertical resource usage in the future.

There is an additional concern - application servers which must remain aware of each other - which we will return to after a discussion of the necessary remaining components of the elastic toolset.

## Getting the Infrastructure Ready

For the purposes of this discussion, I will assume that the reader is functioning in a “cloud”-type virtualized environment. It is possible to scale elastically in a hardware environment, but the complexity level is much higher. While implementing this system, I was working with an internal cloud built on VMware vSphere, with Infoblox providing DNS and DHCP and Cobbler for provisioning and repository management.

The key infrastructure elements needed to support this are as follows:

- Network support - your network devices must support servers being brought up in a variety of subnets. In a virtualized environment, this typically means that the appropriate networks are available to the virtual switches used for provisioning. Depending on the size of your environment and complexity of your network layout, you may need to do additional work on the virtual switch side and VM controller configurations to ensure that new servers are brought up on servers with access to the appropriate subnets. At Advance, where nearly all subnets were available to all VMs for provisioning, this was vastly simplified; in most organizations however this is not the case.
- Network service support - either pre-provisioned static IP addresses for new servers with appropriate ports provisioned, or DHCP. Since most bare-metal configuration requires DHCP and PXE booting capability, having both will make your life much easier. If a subnet fills up, your auto-deployment tools should be robust enough to capture and handle that error, even if only by paging an admin to resolve the problem. One of the reasons the Infoblox was terrific for this deployment was the ease of access to its DHCP interface for both querying of available addresses and provisioning of reserved addresses.
- DNS readiness for automated deployment. This means that your DNS zones should be laid out clearly, with reasonable reverse-mapping of IP addresses, so that automated provisioning is straightforward. The system needs to know what IP address to assign based on system role.
- Appropriate connectivity to build environments. You must have the bandwidth to push down OS images and patch data to multiple servers quickly.

- API or command-line access to your virtualization platform which will enable you to create new VMs, grab their MAC addresses, and hand information about them to your bare-metal deployment system. VMware is shaky in this regard, but it provided enough access for us to comfortably do what we needed.
- Automated OS licensing. If you need to enter a username and password at the console and that information can't be stored in an answer file, elastic expansion is a no-go.
- Automated patch management. This is often overlooked, but it's very important that a server brought up today look like one that was brought up last week. If we install an OS, even from the same image, but then run an update against current package repositories, our server today may have a very different set of packages from the server deployed last week. So it is important that all servers talk to the same repository set, with the same package version information across the board. We were struggling with this when I departed Advance, but had identified the Pulp project as a possible solution.

## OS and Application Deployment

Your OS deployment choices will be largely shaped by your OS choice. As a CentOS environment, we used Cobbler for system deployments. There are a multitude of alternatives - Foreman, Spacewalk, or even hosting kickstart files on a regular webserver. The important thing is that the deployment system be able to identify a host and hand it the appropriate base configuration. Your OS install should be generic and minimal; don't try to handle 50 gold master images, but rather let your configuration management tool handle the heavy lifting.

At Advance, we chose Puppet as a configuration management system, and as I have since left Advance to work for Puppet Labs, my preferences are clear. However using any tool in this space puts your organization light years ahead of most of its competition. The key is not which configuration management tool you use, but the discipline to stick with that tool and keep everything in configuration management. Remember that, as discussed earlier, if it's not in configuration management, it can't be deployed automatically.

At this point I will return briefly to the concept of clusters that are not a collection of naive servers, but which must be aware of their own configuration or of each other. Configuration management provides the solution for this. Servers can be assigned environments or variables based on their intended role or position in a cluster, and configuration files can be templatized based on that information. In Puppet, we can use Exported Resources to ship dynamic information out of nodes to a shared datastore, so that other nodes can learn about them and make decisions. With proper scripting and policies, we can repartition our data sets in what is now a self-aware, elastically growing cluster.

## Ad Hoc Administration

There are circumstances in any IT environment that don't fit well into the paradigm of change/configuration management. Suppose we want to kick all the Apache servers in a particular datacenter, or remount NFS volumes attached to a storage device that went belly-up?

The old solutions were SSH in a for loop, and ClusterSSH, which displays multiple terminals and allows a user to control them all simultaneously. Newer tools in this space provide more accountability and control and better reporting.

At Advance we were using the Marionette Collective, or MCollective, for a few months when Puppet Labs acquired it, cementing our choice. Whether you using MCollective, func, fabric, Knife, or any other tool the important thing is that ad hoc administration should be compatible with your change management environment. If changes in one disrupt the other, automation will break. Many of these ad hoc tools force you into writing clients or carefully-wrapped agent scripts, something seen as an inconvenience. But there's a reason for this: we want to be able to execute something in a controlled period of time and then aggregate and return the results in a meaningful way. We can then store and report on the results and even audit the activities of the people using the tools.

The more centralized and automated this solution, the less likely it is to have unexpected impact on the managed environment. If we take the SSH in a for loop example - if we run that loop against 1500 servers, who is going to parse the results to notice that server 650's response didn't quite look right? And if it didn't, will the next round of changes cause server 650 to diverge even further from the remaining 1499? Tools with built-in auditing and data summarization can find these issues before they become problems or unexplained application behavior.

## Where To Next?

I was saddened to leave Advance before we actually went elastic in production, but we had all the groundwork in place, thanks to the work of our infrastructure team's construction of their vDeploy tool, which interfaced with our VMware, DNS and DHCP environments to deploy new servers, then handed off to my Operations team's Cobbler and Puppet environments.

The workflow was that our Nagios-based monitoring system would trigger vDeploy only if the appropriate business criteria were met, causing vDeploy to build a new host based on information passed from Nagios. The concept of doing this sounded unthinkable at the start of the design process, but after analyzing the problem, it became clear that technologically, there were very few hurdles. Most applications and environments have APIs or RESTful interfaces that can be used for this sort of communication, and writing these scripts was simply a matter of putting in the work.

The actual complexity lay in building the application and business rules around when these things should happen. Focusing on communication and shared information rather than the engineering details proved to be the key. Good engineering and technology selection is key but is made much easier by taking the time to understand the business logic that these engineering exercises are designed to satisfy. While the impulse of many engineers is to jump in and start coding, taking the time to understand and manage the underlying cultural and infrastructure issues can turn development of an elastic environment from a seemingly insurmountable series of roadblocks to an exercise in small-scale script development.