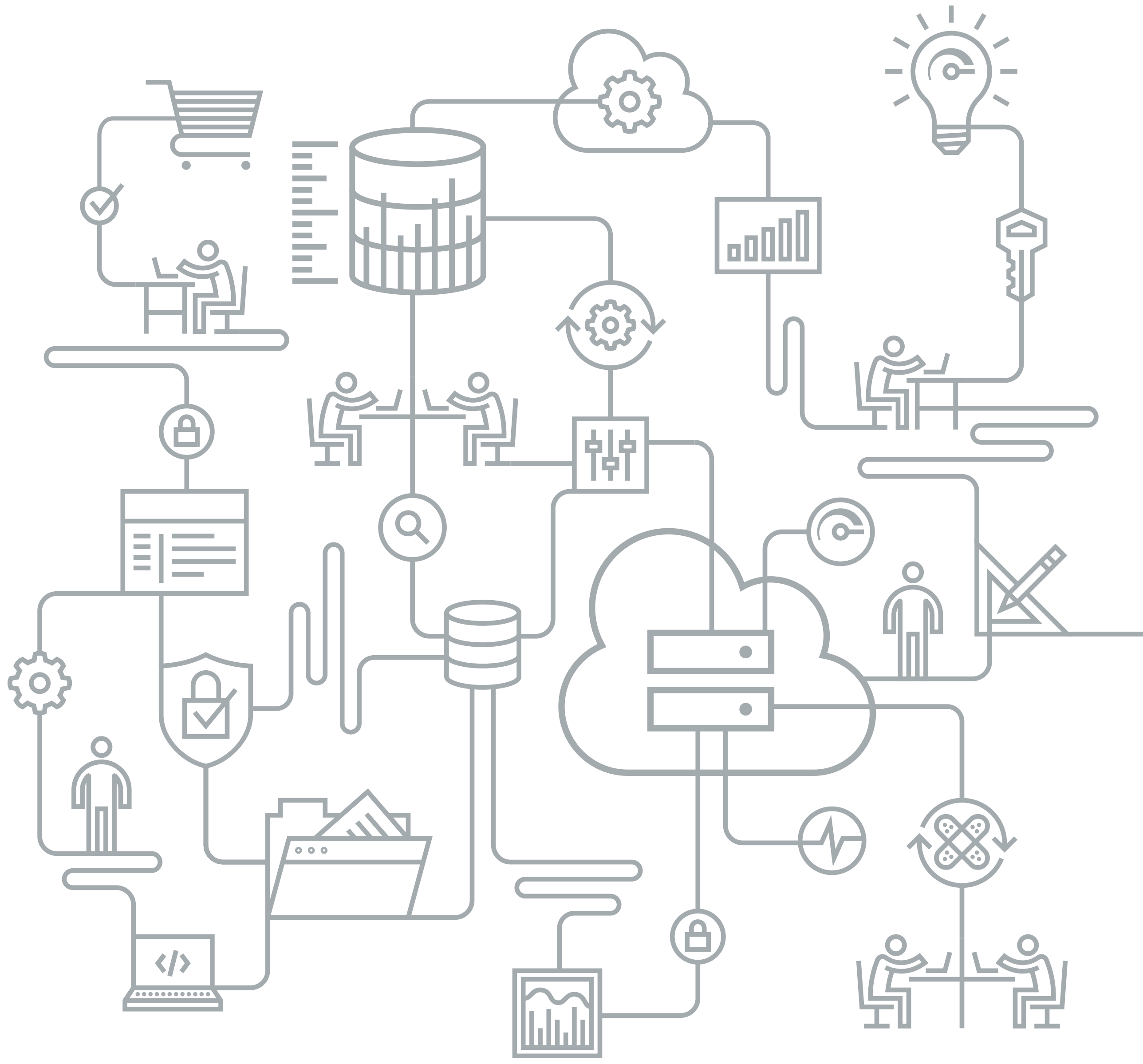
CREATING THE PERFECT AEM/CQ DEPLOYMENT

BEST PRACTICES FROM A LEADING RACKSPACE AEM EXPERT



A white paper by:

TAD REEVES

TABLE OF CONTENTS

INTRODUCTION

System administrators have always loved to blame their development teams for outages. "Everything worked great until you deployed that new code," they say. And developers have always loved to blame system administrators for "slow servers." This kind of finger-pointing after outages is a time-honored tradition, but it's bad for business. It leads to counterproductive outcomes, like admins trying to increase job security by working to impede new code releases, services or technologies. Helping the business roll out cool new features and products becomes secondary.

Yet the whole purpose of having a web infrastructure is to enable firms to problem-solve and innovate to better serve the business. If the business can't roll out new features and systems, what's the website for? What's the point of even having it up?

That's why it's so important to implement a deployment process that is flexible enough to support change. In this paper, I offer guidelines on how to create such a process, so you can build an infrastructure that supports IT and business change.

THE PERILS OF MANUAL DEPLOYMENT STEPS

Over the course of my career, I've had to create, update, take over, rewrite and deal with the eccentricities of many different AEM/CQ website deployments. They've ranged from super-sexy, single-button deployments to utterly horrific manual deployments taking eight-plus hours and involving the manual execution of five-plus pages of Wiki article steps. Theoretically, deploying new code should be a small fraction of the overall process of running a website. In practice, however, I've seen it consume up to 75 percent of the effort of my operations teams, including planning, execution and cleanup.

I credit the amazing John Allspaw, formerly of Flickr, for opening my eyes to the widespread organizational destructiveness of messy, manual deployment processes. He describes the traditional job of systems administrators as "setting up servers that people can use, and which don't go down." But as he so intelligently pointed out seven years ago, the mission of ops guys should be to "continually create and hone a platform that enables the business to do awesome things."

If deploying new code is a trouble-free, low-risk, low-friction activity that can be carried out by product managers and QA folks with extremely limited infrastructure skill — well, that would allow them to focus on new features and general platform awesomeness.

Over the years, Adobe Experience Manager (AEM/CQ/Day Communique/etc.) has been just as brittle and resistant to smooth code deployments as anything else out there. My first CQ deployment involved several relatively brittle Apache Ant tasks to build and deploy the code, and then a variety of manual steps to clear cache, restart servers (usually multiple times), deal with Akamai and futz around with memory leaks, server performance bottlenecks and sundry other issues until things worked right. Deployments consisted of a 10–15 person conference call during off hours and lasted at least eight hours, and sometimes longer. Rollbacks and redeployments were common, as bugs that hadn't surfaced in lower environments were very frequently discovered in production. It was a nightmare. Hence the need for a better process to minimize re-work and speed up necessary builds and changes.

DEFINING AN IDEAL DEPLOYMENT PROCESS

There's an "existing scene" and an "ideal scene." How do we define the ideal scene for code deployment? Everyone has an opinion about this, but in my experience the overarching goal of an ideal deployment process is:

Make software deployment and configuration changes a frictionless, terror-free process, thereby enabling the business to rapidly make changes and improvements to the system, which advances business goals.

More specifically, ensure that:

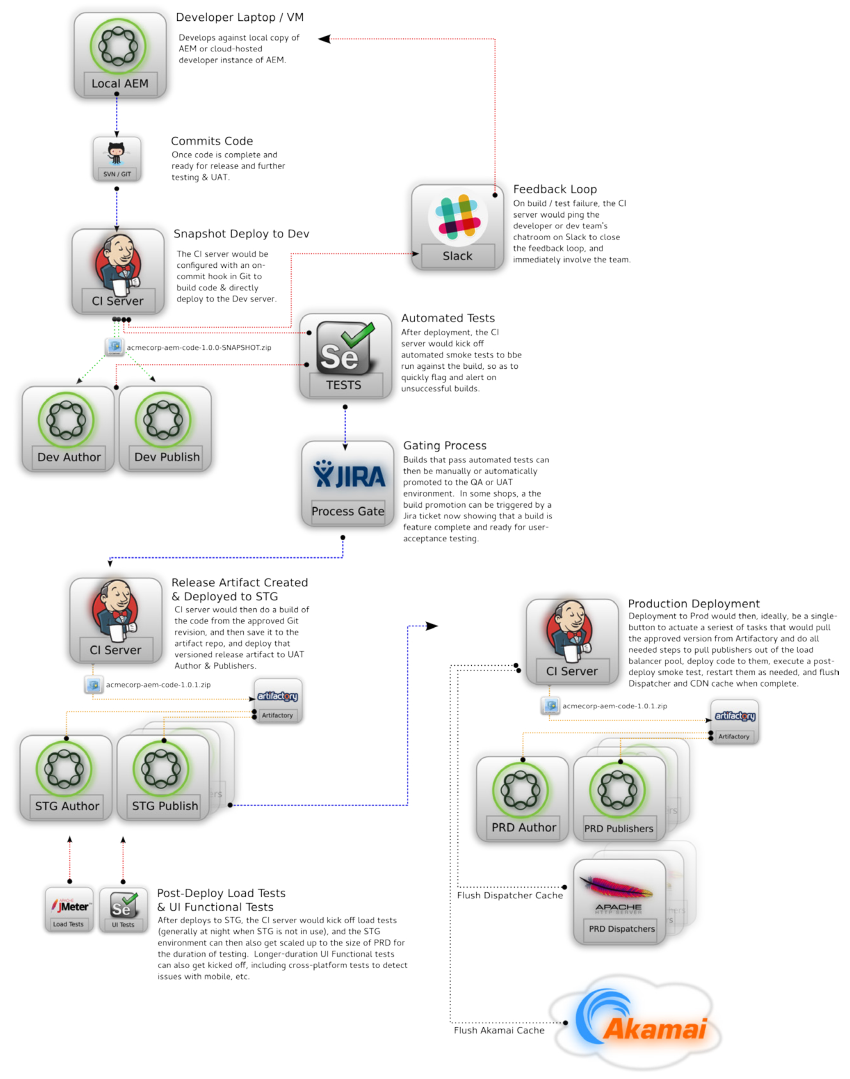
- **Product development can run the code deployment.** Infrastructure doesn't create the site features. Put full control of the deployment into the hands of product and site development staff who understand the features being deployed.
- **The deployment is as low-risk as possible.** Isolate the code rollout process such that all high-impact issues can be caught before the code hits the general public.
- **Rollout and rollback are automatic (or, minimally, require the press of a single button).** Execution doesn't require an infrastructure engineer or a series of complicated manual steps.
- **There are no surprises.** All code and configuration has been rolled out to earlier environments before hitting production. There should be no surprises and no cases of "We can only test xyz feature on production." Everything should be testable before production.
- **Deployments require as few manual steps as possible.** Ideally, all steps should take place within a continuous integration server — no steps directly on the console or on separate tools and pages.

VISUALIZING AN AEM RELEASE PROCESS

The diagram and outline below detail a process that works well with various AEM sites. Both should be useful for understanding the overall process and for fully appreciating the second half of this article, which describes a "perfect" AEM deployment pipeline.

An Example AEM Release Process

This is an example development process. For the purposes of illustration, the process is using Git for version control and Jenkins as CI server, though the same process would work regardless of product choice on each.



Developer Laptop / VM
Develops against local copy of AEM or cloud-hosted developer instance of AEM.

Commits Code
Once code is complete and ready for release and further testing & UAT.

Feedback Loop
On build / test failure, the CI server would ping the developer or dev team's chatroom on Slack to close the feedback loop, and immediately involve the team.

Snapshot Deploy to Dev
The CI server would be configured with an on-commit hook in Git to build code & directly deploy to the Dev server.

Automated Tests
After deployment, the CI server would kick off automated smoke tests to bbe run against the build, so as to quickly flag and alert on unsuccessful builds.

Gating Process
Builds that pass automated tests can then be manually or automatically promoted to the QA or UAT environment. In some shops, a the build promotion can be triggered by a Jira ticket now showing that a build is feature complete and ready for user-acceptance testing.

Release Artifact Created & Deployed to STG
CI server would then do a build of the code from the approved Git revision, and then save it to the artifact repo, and deploy that versioned release artifact to UAT Author & Publishers.

Production Deployment
Deployment to Prod would then, ideally, be a single-button to actuate a seriest of tasks that would pull the approved version from Artifactory and do all needed steps to pull publishers out of the load balancer pool, deploy code to them, execute a post-deploy smoke test, restart them as needed, and flush Dispatcher and CDN cache when complete.

Post-Deploy Load Tests & UI Functional Tests
After deploys to STG, the CI server would kick off load tests (generally at night when STG is not in use), and the STG environment can then also get scaled up to the size of PRD for the duration of testing. Longer-duration UI Functional tests can also get kicked off, including cross-platform tests to detect issues with mobile, etc.

To detail the process, the release would consist of:

1. **Local Development Work:** After getting a feature, the developer can work on his own local/personal copy of AEM. Depending on your app and the DevOps resources you have, this could be running directly on the developer's laptop; located in a Vagrant-deployed, centrally managed VirtualBox container that he checks out daily; or located on individual cloud AEM deployments that are generated for each developer or development team. Some development teams use AEM instances hosted in the Rackspace Public Cloud or AWS, which can be powered on during development and then shared with the team to facilitate peer reviews or pre-commit UAT to make feedback on features even faster. Regardless of location, however, the developer should work on features on this dev environment, and should not commit code until features are complete and ready for testing or UAT.

2. **Shared Development/CI Environment:** After code is committed to the repository, the continuous integration (CI) server should have a post-commit hook to check out and deploy this code to a shared dev environment. It should execute a full SNAPSHOT build of the code, create a code package (named something like acmecorp-aem-code-1.0.0-SNAPSHOT.zip) and deploy it via the AEM Package Manager on the dev author and publisher servers.

3. **Post-Deploy Automated/Manual Testing:** Ideally, you will then have automated testing fire-off as a part of the build, in order to have an immediate feedback loop regarding the quality of the commit. This automated testing will (ideally) smoke-test the application and give immediate feedback (generally via a chat tool like Slack/Hipchat).

4. **Release Candidate Gating:** Generally, the only ones that get deployed to are dev servers when someone commits code. Past dev teams generally implement a gating process – preferably automated, but usually manual – where a release candidate is identified and promoted. You either tag this in version control or simply say, "The build, as it exists right now in revision 29560, is a release candidate." At that point, the individual in charge of gating can promote the build.

5. **Build and Deploy a Release Artifact to QA/STG:** Next, you have your CI server conduct another build, but instead of releasing it directly to your QA or Stage environments, you version and release it to an artifact repository. An artifact repository is a class of software (e.g., Sonatype Nexus, Jfrog Artifactory and Apache Archiva) that stores versioned binary artifacts and provides control over dependency management. This article gives a great overview of the purpose of an artifact repo. Once you've built and saved a release artifact of your AEM code package – let's call it acmecorp-aem-code-1.0.0.zip – your CI server downloads it from the artifact repo and deploys it onto QA or STG for testing. Use of the artifact repository can then ensure that you deploy identical code to all servers, and that the "1.0.0" code package you deploy will contain exactly the same code that you release to production, once tested.

6. **Automated Testing on Staging Environment:** Generally, you then fire off a process to execute longer-running automated tests on your staging environment. Ideally, this includes functional testing of the software and its key integration points as well as load testing to both validate speed optimizations and verify that new features don't create performance degradation or server instability under load.

7. **Production Deployment:** Assuming successful completion of the automated test suite, as well as passing the other automatic or manual gating process you have in place, you then execute a production deployment. Deploying to production generally includes a few key processes:

- **Alerting:** Calls to your monitoring software to pause alerting during your deployment window, so that your service desk (or Rackspace support team) are not inundated with false-positive alerts.
- **Load Balancer:** Interaction with your load balancer to take individual nodes out of the pool during deployment. How this is done will depend on your AEM architecture, whether or not your publishers and dispatchers are lined up 1:1 or are each behind their own load balancers, etc. Regardless, even though AEM can have code deployed to it while it's hot, you will not want to have your servers live and taking traffic during a deployment. There will always be a window during code installation when the server will be responding with errors and may crash if under a heavy load. You'll want to ensure that each node is taking no traffic while it's being deployed to.
- **Deployment:** The act of deployment here should last approximately 30 seconds per server, as the only activity will be downloading the designated version of your code out of the artifact repository and installing it using AEM Package Manager's web service interface – an activity that generally only takes a few seconds.
- **Restarts:** Depending on your code, about 50 percent of AEM sites I've seen require the AEM service to be restarted post-deployment (and sometimes pre-deployment as well) in order to respond consistently and stably. Server restarts can generally be accomplished automatically using your CI server (Jenkins, Team City, etc.) or an orchestration-tier product such as Rundeck or Ansible Tower.

8. **Monitoring & Dashboarding:** The most successful AEM sites have a simple maxim with respect to monitoring: *We should be able to clearly display on a dashboard any major factor that could affect the site's performance, stability or availability.*

I could write a whole book on this. Your site should have dashboards that you've created using log aggregation software (Splunk, Loggly, Sumo Logic, etc.) and application performance monitoring (APM) tools like New Relic or AppDynamics. Immediately after the deployment, and while your QA team is conducting post-deployment validation, you should monitor these dashboards closely for changes in response time, CPU load, disk utilization, cache-hit ratio, etc. I've found Geckoboard to be a great dashboard aggregator, able to combine disparate data streams in a single-pane-of-glass view. However you construct it, such a dashboard is essential, as it can immediately show leading indicators of degraded performance or failure before outages or compromised functionality occur.

9. **Rollback:** When you need to roll back, your code artifact versioning becomes extremely important. Let's say your release just installed acmecorp-aem-code-1.0.1.zip to replace the previous version, 1.0.0. When you installed 1.0.1, AEM Package Manager automatically deactivated and deprecated version 1.0.0. If you realize that 1.0.1 is tragically flawed and has some previously uncaught bug, rolling back code is as simple as removing the 1.0.1 package from AEM Package Manager. Version 1.0.0 will then be re-installed automatically, and the site will immediately be up and running on the older version of the code.

COMPONENTS OF A PERFECT AEM DEPLOYMENT PIPELINE

A number of technologies and practices can help create a more reliable, repeatable and terror-free deployment process. Some are low-effort, high-payoff items, while others are much more complex to implement and offer a lower return in terms of work reduction and benefit. I have sorted this list so that the most important, "Don't think about not doing this" items are at the top, followed by more optional and/or potentially controversial topics.

1. Version Control

- **Version your code:** Hopefully, if your project is on or moving to a complex platform like Adobe Experience Manager, your team's codebase is already version-controlled. Otherwise, there is zero option for you — pick a version control platform (Git, Subversion, Mercurial, Microsoft Team Foundation, etc.) and get everyone using it.
- **Version your server configs:** Get all of your key infrastructure configs into version control, even if you don't deploy them automatically. Just saving your configs out to version control will give you a place to revert to if something goes awry.

2. Deploying Your Code

- **No manual "hot" configuration changes unless it's on a dev environment:** AEM has a few settings (JVM, repository, etc.) that are set with on-disk configuration files, but the vast majority of AEM's configuration happens in the OSGI console or by direct editing of nodes in CRX/DE. These configs can usually be edited while the server is hot, and the flexibility of doing so can lead developers and engineers into the bad habit of making these changes in the UI, as opposed to in versioned code. Making hot changes to the server opens the door to massive and extremely difficult-to-detect differences both between environments (i.e., DEV/QA/STG/PRD) and between machines in the same environment. An example: Let's say marketing wants a URL rewritten in Sling. A developer then goes in and manually edits the /etc/map entries in CRX-DE to effect the desired change. Once this change is tested on DEV, the developer should commit this change into version control, and have it deployed via a package that installs Sling rewrite maps. That way, he can be certain that all instances up the chain get this same fix, eliminating a possible config difference between servers.
- **Deploy versioned packages:** This is an important and very poorly documented part of the package deployment process. In Apache Maven parlance, "SNAPSHOT" is a special version number that indicates a current development copy of software that isn't yet released and is not ready for release. The idea here is that before a 1.0 release (or any other release) there exists a "1.0-SNAPSHOT" that might become the 1.0 release. The difference between "1.0" and "1.0-SNAPSHOT" is that downloading "1.0-SNAPSHOT" from an artifact repository today might give you a different file than downloading it yesterday or tomorrow. Conversely, the acmecorp-aem-code-1.0.1.zip package will be entirely unique, and even the slightest change to the codebase creates an entirely new version of the code. Once a release candidate has been identified in QA (i.e., QA signs off and says, "v1.0.1 is a pass, is OK to go to prod"), you won't rebuild before deploying to prod but will instead deploy that exact artifact.
- **No individual bundles — deploy only packages:** While it's technically possible to make individual cURL calls out to the AEM OSGI console and individually deploy code bundles, this is 1) outside the package management and versioning process of AEM Package Manager and therefore 2) very difficult to control and track. A healthy percentage of your AEM availability, functionality and performance issues will take place around the time of a deployment, so being able to definitively tell when a server has had code applied to it, and by whom, is critical for the debugging process.

3. Choose a CI Server That Works for You

This point should have its own series of blog posts, but this wiki page is a good place to start. I've seen AEM shops work well with Jenkins/Hudson, Team City by Jetbrains and Thoughtworks Go, though there are many other high-quality solutions.

4. Artifact Creation, Versioning and Promotion

- **Define a process:** Your team will want to define a process for creating, versioning and promoting your release artifacts. As discussed earlier, it's imperative to have a clear difference between your continuously built SNAPSHOT artifacts and your versioned release artifacts.
- **Pick an artifact repository that works with your process:** There are a number of high-quality and free or relatively inexpensive artifact repositories. As detailed earlier, an artifact repository is a class of software like Sonatype Nexus, Jfrog Artifactory and Apache Archiva, which store versioned binary artifacts and provide control over dependency management. This article gives a great overview of the purpose of an artifact repo. Team City and Jenkins also store build artifacts internally, but don't do things like dependency management. Definitely treat this as one of the critical pieces of your deployment stack.

5. Make Stage Look Exactly Like Production

"It worked on STG and all the tests passed. Why did it break prod?!" This conundrum has dogged nearly every team I've been on, and "Stage is not exactly the same as prod" has been the outcome of more root-cause analyses than I care to mention. There are a number of reasons why companies have a hard time with this one. Most authoring only happens in the prod environment, and the prod authors and publishers generally wind up with much more content than any other environment; they also have all of the cruft associated with frequent revisions, backed-out content, expired partner articles and the like. Additionally, many companies opt to cut costs by making their lower environments significantly smaller than prod, creating a load-testing challenge. Fortunately, this critical problem has several solutions:

- **Leverage the cloud to right-size the stage environment:** Most of the time, companies use their staging environments to do final pre-flight checks on content and code before they go live. Ninety percent of the time, stage just needs to be functionally identical to prod. However, imagine a well-trafficked prod environment composed of 16 publishers and 16 dispatchers spread across two geographical regions. For accurate functional testing, you could generally get by with only one or two publisher/dispatcher pairs, but for load testing you would want to be able to test things like mass-replication events (where the author is under heavy stress to replicate to all 16 publishers) and cross-coast load balancing (where you're testing to ensure that load-balancing algorithms effectively balance traffic or accurately geo-place it). For testing of this nature, you want a fully built environment the same size as prod. However, such an environment can be very expensive.

  The solution here is to leverage the power of the cloud to right-size your STG environment. Under normal circumstances, you could keep the STG environment at two publishers/dispatchers to enable user-acceptance testing and UI functional testing, both manual and automatic. However, when the need arises, you can leverage Rackspace's engineers to dynamically scale the environment up to the size of production for a matter of hours or days, as appropriate, so as to accommodate testing, and then scale it back down when testing is completed. That way, you don't pay for equipment you don't need, but still have an environment mechanically identical to prod.

- **Use infrastructure automation to ensure PRD and STG are identically configured:** The more manual configuration in an environment, the more potential that a setting will be missed, creating differences between environments. This is where an automated AEM infrastructure such as the Rackspace Managed Cloud for AEM (MCAEM) can be extremely valuable. AEM's configuration extends far beyond on-disk configuration files and includes literally thousands of switches and levers deep in the OSGI console that can have pervasive and hard-to-detect differences between servers. Using an automated solution like Rackspace MCAEM, you can make an exact clone of the entire production environment (content and all), even for a short period, and execute testing against that environment.

  If you really want to get fancy, you can use a CNAME switch to cut over to this new prod environment clone after testing is completed, and then decommission or deprecate the old prod environment once validation and burn-in are complete.

  But even without a fully automated solution like MCAEM, tools such as Ansible, Chef and Puppet can manage your configurations to ensure that STG really is an exact copy of prod and preclude any unpleasant surprises.

- **Regularly copy prod content to lower environments:** The most common reason issues aren't caught before production is that lower environments aren't loaded with full production content. Menus, full-text searches, search-driven pages, etc. all depend on fully populated content to display properly, and features that leverage them are impossible to fully validate without complete, up-to-date content.

There are multiple ways to get production content down to lower environments, and the ones you pick will depend on many factors, including content size, site infrastructure and level of regular flux. Some sites can get by with lower-environment content refreshes every six months or so. Others need daily refreshes or development will run into problems. Generally, the solutions at your disposal are:

- **Sync the whole environment:** You may be able to simply take a snapshot of your production author/publisher and copy it to STG or dev and have dev mount the copy. This works especially well in the cloud, where Rackspace block storage or Amazon EBS volumes can be moved and copied easily from instance to instance.
- **Package your content:** For smaller sites, your content can be packaged up using the CRX Package Manager (or automated using REST calls) and then loaded onto lower environments. This method generally becomes unwieldy if more than a few gigs of content at a time are being moved around, as package installation can become unreliable.
- **Use Vault (vlt) to do server-to-server copies:** The Vault tool can be used to do direct server-to-server copies of content, and can be scripted to sync PRD and STG environments on a regular basis.
- **Third-party tools:** Third-party tools such as Recap can also be leveraged, but some of them are finicky in terms of which OS libraries are required to support them.

6. Post-Deploy Review and Dashboards

Critical to a successful deployment process is adequate data to determine whether new code is a success or failure. I've commonly seen manual UI functional testing as the only post-deployment QA done on a website. Yes, it's important to verify important functionality manually, but errors often occur under the hood, and the only way to ensure application health is with detailed log analysis and application performance monitoring (APM) tools.

Story time: I once helped a client launch a large, content-driven website that had recently migrated to AEM. The site launched with an auto-complete search feature, which allowed users to rapidly search hundreds of thousands of content bits. The auto-complete function worked, and the website appeared (via the browser) to be performing well. However, leading indicators from the before/after of the code release showed that not all was well — high CPU despite low numbers of actual page requests and a search subsystem nearly pummeled into the ground. Investigation via Splunk rapidly revealed that the auto-complete search feature was recursively executing a full-text search 10 times every time a user typed another character in the search box. A user searching for "barbecue" would create 80 individual search requests — and many more if he or she forgot how to spell and kept backspacing.

The moral of the story is that this issue would have taken down the site had the load been heavy — and the problem was identified in time only because we'd created dashboards with all the leading site-performance and error indicators visible and current.

- **What you want on your dashboard:** There are a number of leading indicators you'll want on your information radiators or DevOps dashboards for AEM, depending on the features you use. They will be somewhat different for every installation. The following are the ones I've found most useful:

1. **Publisher CPU%**
2. **Publisher disk I/O**
3. **Publisher disk utilization**
4. **Publisher requests/sec.**
5. **Publisher error rate**
6. **Author activations**
7. **Dispatcher cache-hit ratio** (divide dispatcher requests by publisher requests)
8. **Dispatcher disk utilization**
9. **Dispatcher worker thread status**
10. **Cache invalidation requests**
11. **Search head hits** (for sites with Solr/Endeca/etc. for search)
12. **Author CPU%**
13. **Authors logged in**
14. **Author workflows running**
15. **Author disk I/O**
16. **Author disk space**
17. **Author error rate**
18. **Maintenance tasks** (TarMK optimization, datastore garbage collection (AEM 5.6) or compaction (AEM 6.x), backups, etc., as they can dramatically affect performance)
19. **Import/export status** (many sites have regular feeds that import into AEM in batches, and when run can cause major performance hits)

- **Acquire and use an APM tool:** With a platform the size and complexity of AEM, it's imperative that you have a tool that can quickly supply information on performance and functionality issues. APM suites such as New Relic and AppDynamics are our preferred tools at Rackspace, used by our Critical Application Support (CAS) teams to detect and handle leading indicators of performance and availability issues. However, we also train our clients' development teams to use them — if performance issues can be detected and handled on lower environments by dev teams, they don't have to become an operations problem.
- **Acquire and use a log aggregation and analysis tool:** If your logs aren't already being ingested into a tool like Splunk, Sumo Logic, Loggly, etc., you'll want to use one. The majority of the dashboard items mentioned above will be pulled from your AEM error and replication logs, so you'll never be able to visualize the status of your environment without such a tool. Also, the more your site is multi-server and multi-region, the less you'll be able to get a picture of what is happening with your application by tailing the log of a single server.

As a final plug, the most influential factor I've seen in getting dev and ops teams to work together is shared use of log aggregation tools like Splunk or Sumo Logic. Being able to quickly highlight and collectively share time coincidences, error messages and other such visual and conclusively telling data quickly eliminates contention and "throwing it over the wall." Instead, it allows teams to rapidly work together to resolve issues.

7. Eliminating Manual Steps

Executing a deployment with minimal IT-ops interaction means eliminating manual steps in the deployment process. The major ones that need to be handled on the CI server or in your deployment pipeline are:

- **Load balancer handling:** Although you can deploy code to an AEM server while the server is hot, it's extremely unwise to do so while the server is under load of any sort and taking public traffic. There will inevitably be a point during deployment, as old code is being replaced, when the server will throw errors to end users. If the server is under heavy load when the deployment is happening, it could crash altogether. Therefore, it's important to be able to take each publisher out of your load-balanced pool during the deployment process, and preferably while it's being manually verified or automatically smoke-tested. This means that you will want a simple, bulletproof way to take a publisher out of the load balancer.

  Some companies do this with an API call to their F5 load balancer (if they're on hardware) or with an API call to AWS, Rackspace or Azure cloud load balancers. Another method I've seen work very well is one of the oldest tricks in the book — simply code a very lightweight page that can be served from the publisher with a text string (e.g., "publisher1 IN SERVICE") that your load balancer is looking for to determine if it is healthy; if you want to take that publisher out, just change the text string and the publisher stops getting traffic.
- **Dispatcher flushes:** After deployment is complete, most teams flush the dispatcher cache. The dispatcher's cache invalidation process is generally good at keeping cache fresh, but the introduction of new code during deployment changes how pages are rendered, and that change doesn't then go and invalidate pages and/or cached CSS/JS assets. So inserting automation to flush the dispatcher cache — either with scripted Linux "rm" commands or with the ACS Commons Dispatcher Flush UI — will need to be handled in your prod CI server's deployment routine.

  Important note: Flushing dispatcher cache may be something you need to observe the effects of and iterate on, especially for sites with heavy traffic or heavy reliance on cached pages that take a long time to render un-cached. You may want to pre-warm your dispatcher cache with your 20 most frequently hit pages before you put your dispatchers back into the load balancer pool, in order to avoid excessive and potentially crippling load on the publish tier.
- **Akamai Cache Flushes:** Cache flushes of your CDN can generally be done with an API call, although flushing all edge servers typically takes quite a while, sometimes over an hour.
- **AEM restarts:** It's been mentioned before, but with many AEM sites it's essential for reliability to recycle AEM after every code deployment. You will want to set up your CI server to SSH into individual publish instances to execute the restarts and alert you to failed/hung restarts.
- **Pushing dependent/related dispatcher changes:** Often, dispatcher filter rules or Apache rewrites go along with a major code release and have to be done alongside the code release for functionality to work. While you will want to automate this, it gets us into our next point, which involves configuration management.

8. Deploying with or in concert with Chef/Puppet/Ansible

Any shop with more than a handful of servers will eventually need to clear the hurdle of choosing and implementing a configuration management framework such as Chef, Puppet or Ansible. I'm not going to discuss that issue here, however, as it spans a much broader and extremely important topic — how you're building your complete environments, not just the code that runs on them.

CONCLUSION

Adobe Experience Manager is a powerful and complex application that acts as the digital front for your clients. With the growing importance of AEM as a centerpiece of various mobile app, mobile site and desktop channels, your AEM deployment is more important (and probably more difficult) than ever. By employing the best practices outlined here, you can minimize problems and cut down on mistakes so that your AEM application goes into production more easily. No more blame games between devs and system admins.

If you'd like to discuss your own infrastructure challenges, and how a multi-cloud partner like Rackspace with deep AEM experience can improve your process and infrastructure, please contact us. We are more than happy to chat with you and provide a free consult. Contact us today at 1-800-961-2888 or learn more at www.rackspace.com/digital/adobe-experience-manager.

About Tad Reeves

Tad Reeves is an Adobe Experience Manager engineer on the Rackspace Critical Application Support team. He came to Rackspace after working as a DevOps engineer on numerous AEM/CQ-driven websites, and he's worn a number of other hats related to site development, UX design, infrastructure architecture and dreaming of the perfect deployment pipeline. He lives in Portland, Oregon, and spends as much time as possible mountain biking with his wife and three kids. You can read more of his work here:
http://blog.rackspace.com/author/tadreeves.

ABOUT RACKSPACE

Rackspace (NYSE: RAX), the #1 managed cloud company, helps businesses tap the power of cloud computing without the complexity and cost of managing it on their own. Rackspace engineers deliver specialized expertise, easy-to-use tools, and Fanatical Support® for leading technologies developed by AWS, Google, Microsoft, OpenStack, VMware and others. The company serves customers in 120 countries, including more than half of the FORTUNE 100. Rackspace was named a leader in the 2015 Gartner Magic Quadrant for Cloud-Enabled Managed Hosting, and has been honored by Fortune, Forbes, and others as one of the best companies to work for.

Learn more at www.rackspace.com or call us at **1-800-961-2888**.

rackspace®

NOVEMBER 14, 2016 | DIG-CWP-Creating_the_Perfect_AEM/CQ_Deployment-4155-v01