Automating Software Deployment:

Continuous Integration, Delivery, and Deployment

Adeeb Ahmed

Illinois Institute of Technology

April 2017

Abstract

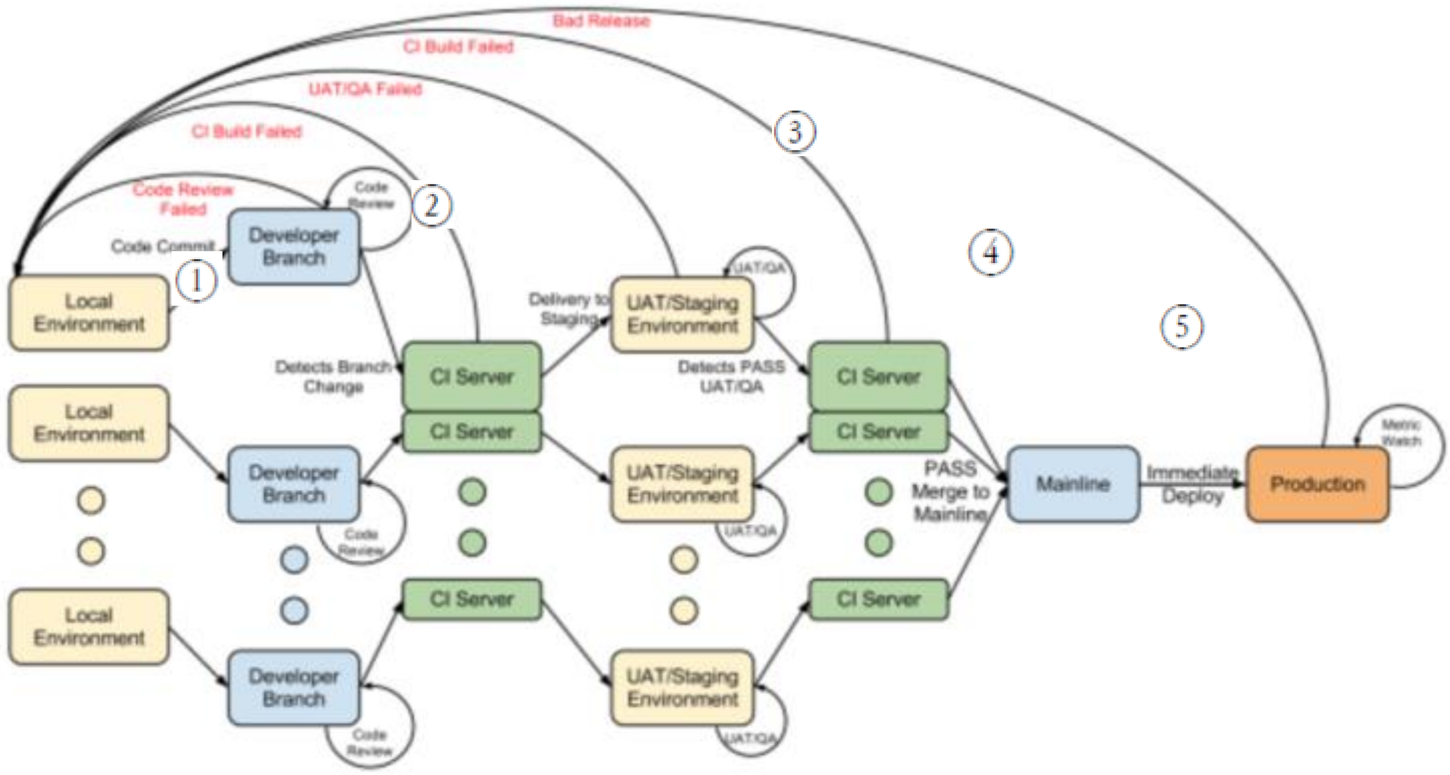
Deploying software is a tedious task, especially for companies neglecting to use the continuous integration tools offered today. Tools like Jenkins help automate the software deployment process and prevent unnecessary technical debt. Without proper deployment, engineers will spend massive amounts of time building a complex system, only to see it crash and burn come deployment. The development of an automated deployment system is a very complex problem. Identifying and comparing the different methodologies of automated software deployment will provide valuable insight about software development life cycle.

The “poster child” of software deployment failure belongs to company by the name of Knight Capital Group (KCG). In 2012 Knight was the largest trader in US equities with market share of around 17% on each the NYSE and NASDAQ. That is until, a failed deployment caused the mother of all bugs. Knight Capital Group went from being one of the largest traders in the US equites market to bankrupt in a mere 45 minutes. Due to the structure of the company’s technology “there was no kill-switch (and no documented procedures for how to react) so they were left trying to diagnose the issue in a live trading environment where 8 million shares were being traded every minute” (1). This disaster was caused by a human mistake in a fully *manual* deployment process. However, the engineer who deployed the bug is not solely to blame. The real fault lies in KCG’s deployment process, which was not appropriate for the risk they were exposed to.

A robust automated deployment process consists of 3 main steps: Continuous Integration (CI), Continuous Delivery (CD), Continuous Deployment (CDL). “Continuous integration is the practice of constantly merging development work with a Master/Trunk/Mainline branch so that you can test changes and test that those changes work with other changes” (2). “Continuous delivery is the continual delivery of code to an environment once the developer feels the code is ready to ship - this could be UAT, staging or production” (2). “Continuous deployment is the deployment or release of code to production as soon as it’s ready. There is no large batching in staging nor a long UAT process before production” (2).

In an ideal workflow the process of deployment should be automated from start to finish such that scenarios like KCG cannot occur. To implement this process requires several servers working together simultaneously and tightly coupled system of checks and balances. The following diagram describes the components of a mostly-automated deployment system:

Process Diagram 1: Assembla Implementation (2)

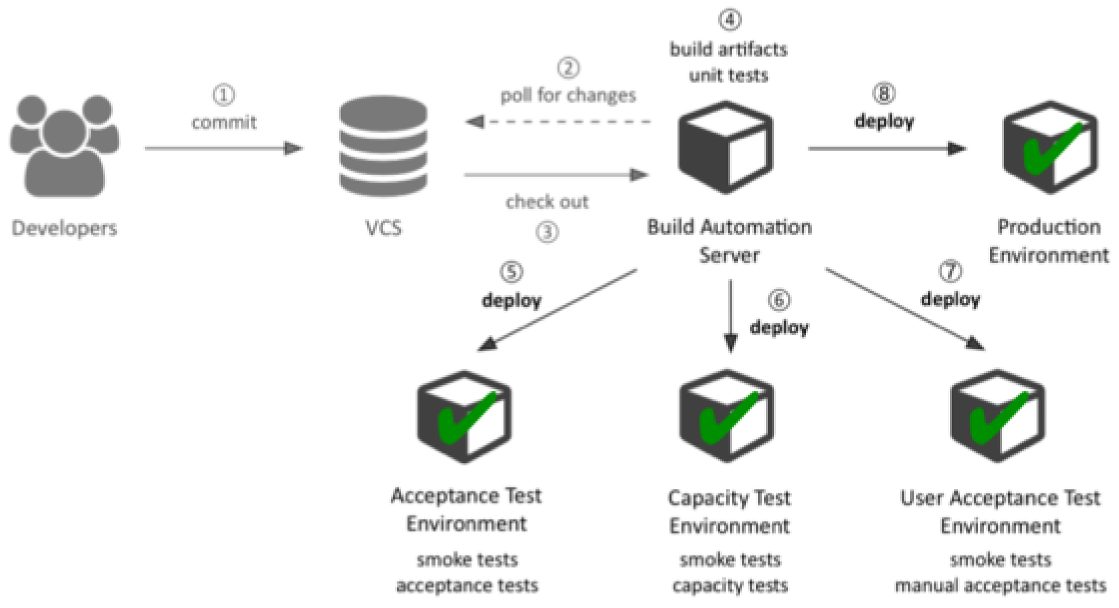


This automated deployment workflow involves 5 semi-automated steps to ensure a successful deployment:

1. Developer checks in their code from their Local Environment into a dev branch.
    a. Another dev on the team can review the code before it’s sent to CI server
2. The CI server then merges the changes to a Master branch, and performs unit tests.
3. If the CI server votes to merge it to the Stage Environment, then the developer can deploy to the Stage and from the QA engineer will test the environment.
4. If all is well then another CI server will determine whether the code is sufficient enough to be merged to the Mainline (master branch).
5. After the code is merged to the Mainline it will be immediately deployed to the production environment.

This process can vary from organization to organization, based on their needs and requirements. More specifically this type of automated deployment pipeline would be used to deploy very large and critical enterprise systems. It’s a combination of both automation and manual deployment to ensure critical systems are deployed properly. To mitigate risk, Knight Capital group should have adopted a model similar to process diagram 1.

Process Diagram 2: Dynatrace Implementation (3)


Developers
① commit
VCS
② poll for changes
③ check out
④ build artifacts unit tests
Build Automation Server
⑧ deploy
Production Environment
⑤ deploy
⑥ deploy
⑦ deploy
Acceptance Test Environment
smoke tests acceptance tests
Capacity Test Environment
smoke tests capacity tests
User Acceptance Test Environment
smoke tests manual acceptance tests

Process diagram 2 illustrates another implementation of an automated deployment system. This implementation in particular is more geared towards user applications and is much more automated and quicker than in process diagram 1. The build automation server acts as the pipeline’s control center and can be implemented with a custom solution or proprietary solution like Jenkins or Ansible. When the build automation server” observes a change in the repository, it triggers a sequence of stages which exercise a build from different angles via automated tests, but terminates immediately in case of failure” (3).

It takes a fully-automated 8-step approach to fully automate the deployment process:

1. Developer commits code to their version control system (Git, Subversion, etc.)
2. Then the build automation server polls for changes to the version control system
3. From there the build automation server checks out the software repository into a temporary directory.
4. Then the build automation server compiles the code and executes any quick running, environment, unit tests, and then release artifacts such as installer packages in addition to generating documentation.
5. Then environment acceptance testing is done by running deployment automation scripts that create an environment that is highly similar, but not necessarily identical, to the production environment.
6. After that, capacity testing is done to ensure the software can maintain its services under production-like load conditions. This step verifies the non-functional requirements.
7. Last stage prior to deployment is user acceptance testing which is usually done on site where the software is being used, and vetted by users for its intended purpose.
8. Finally, after all prior test are passed they software application is ready to be deployed to the production environment.

An example of using this fully automated process would be for customer service related software. For example, say you own a popular retail chain and need a better way to give customer items they have put on hold (layaway system). The solution could take the form of a software application that could be used on a tablet or phone. After getting past all the testing steps (steps 4-6) an engineer would need to go to the store in order to verify the customer service employee can use the software to do the tasks at hand. Though step 7 requires manual intervention it's still considered a fully automated process since every step prior to it requires no human interaction besides writing the tests.

Using process diagram 2's approach to deploying a high volume stock trading application, similar to KCG's, would not be ideal for the risk at hand. There are less human checks which leaves room for failure. While in the case of a retail store, if customer service software is buggy, or even fails, it won't destroy the business in 45 minutes. These are the primary tradeoffs between the 2 approaches to automating deployment.

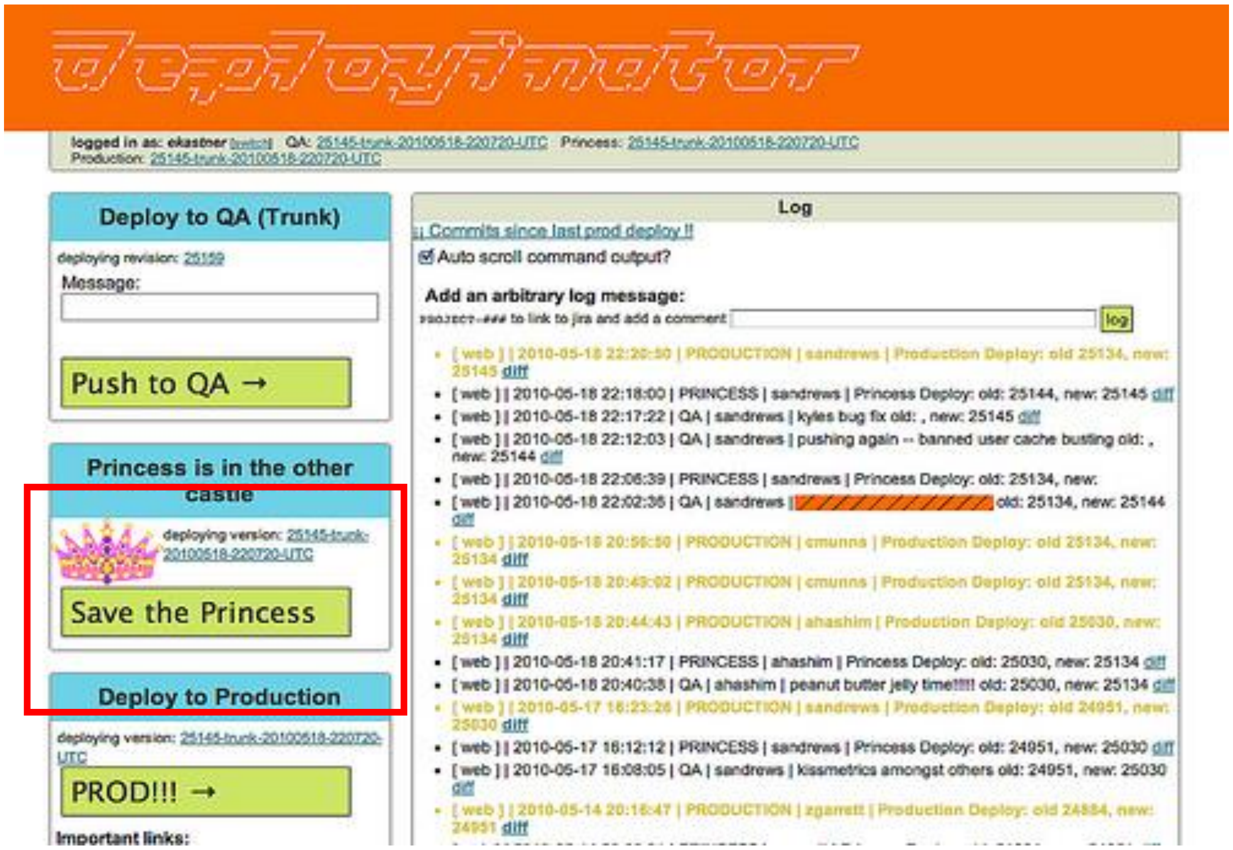
Assembla's and DynaTrace's deployment pipeline infrastructure relies on proprietary CI solutions such as Jenkins and Ansible. However, a NASDAQ listed handmade arts and craft application known as Etsy (NASDAQ:ETSY) has open-sourced their in-house deploying tool known as Deployinator. The ideology driving the tool is heavily based on continuous deployment. If engineers can effortlessly push web code to production, they can then focus more on the needs of the business rather than fighting fires cause by deployment failures. By using Deployinator Etsy engineers have "brought a typical web push from 3 developers, 1 operations engineer, everyone else on standby for over an hour down to 1 person and under 2 minutes" (4). When building Deployinator Etsy engineers did not want to reinvent the wheel (i.e. Jenkins, Ansible, etc.), instead they focused on ease of deployment by narrowing down these key requirements:

- Build for web based application
- Robust logging (Who, What, Where)
- Adaptable to different networks
- Run from a centralized location
- Announced in our IRC and email
- Transparent in regards to its actions
- Integrated with our graphing/monitoring tools

By focusing Deployinator to only web based applications they were able to create a much lighter deployment system compared to robust tools (i.e. Jenkins, Ansible, etc.). With built in logging engineers no long need to integrate logging tools with their application (i.e. Kafka), they can simply rely on Deployinator to log the who, what and where. Since it's a centralized deployment system, rather than being split between many servers, it's much more flexible and adaptable to different networks (not to mention it's open source). Deployinator also is transparent with what's being deployed into the production environment by being able to send out updates via IRC's and email notifications. Lastly, Deployinator comes with graphing and monitoring tools eliminating the needs for third party tool such as Datadog, though you can still use third party tool if you prefer.

One of the core features of Deployinator that separates it from the likes of Jenkins and Ansible is a staging environment known as "Princess".

***Etsy's Deployinator* (4)**



logged in as: ekastner [switch] QA: 25145-trunk-20100518-220720-UTC Princess: 25145-trunk-20100518-220720-UTC
Production: 25145-trunk-20100518-220720-UTC

deploying revision: 25159

deploying version: 25145-trunk-20100518-220720-UTC

deploying version: 25145-trunk-20100518-220720-UTC

¡¡ Commits since last prod deploy !!

☑ Auto scroll command output?

Add an arbitrary log message:
PROJECT-### to link to jira and add a comment

```
• [ web ] | 2010-05-18 22:20:50 | PRODUCTION | sandrews | Production Deploy: old 25134, new: 25145 diff
• [ web ] | 2010-05-18 22:18:00 | PRINCESS | sandrews | Princess Deploy: old: 25144, new: 25145 diff
• [ web ] | 2010-05-18 22:17:22 | QA | sandrews | kyles bug fix old: , new: 25145 diff
• [ web ] | 2010-05-18 22:12:03 | QA | sandrews | pushing again -- banned user cache busting old: , new: 25144 diff
• [ web ] | 2010-05-18 22:06:39 | PRINCESS | sandrews | Princess Deploy: old: 25134, new:
• [ web ] | 2010-05-18 22:02:36 | QA | sandrews | old: 25134, new: 25144 diff
• [ web ] | 2010-05-18 20:56:50 | PRODUCTION | cmunns | Production Deploy: old 25134, new: 25134 diff
• [ web ] | 2010-05-18 20:49:02 | PRODUCTION | cmunns | Production Deploy: old 25134, new: 25134 diff
• [ web ] | 2010-05-18 20:44:43 | PRODUCTION | ahashim | Production Deploy: old 25030, new: 25134 diff
• [ web ] | 2010-05-18 20:41:17 | PRINCESS | ahashim | Princess Deploy: old: 25030, new: 25134 diff
• [ web ] | 2010-05-18 20:40:38 | QA | ahashim | peanut butter jelly time!!!!! old: 25030, new: 25134 diff
• [ web ] | 2010-05-17 16:23:26 | PRODUCTION | sandrews | Production Deploy: old 24951, new: 25030 diff
• [ web ] | 2010-05-17 16:12:12 | PRINCESS | sandrews | Princess Deploy: old: 24951, new: 25030 diff
• [ web ] | 2010-05-17 16:08:05 | QA | sandrews | kissmetrics amongst others old: 24951, new: 25030 diff
• [ web ] | 2010-05-14 20:16:47 | PRODUCTION | zgarrett | Production Deploy: old 24884, new: 24951 diff
```

Like most automated deployment systems Deployinator had a “Staging environment” which is the last stop before code went live in the real world. The problem with that is most staging environments are highly similar to the production environment, but not necessarily identical. This is a problem because the first time your code interacts with the actual production environment would be when you deploy. It becomes very difficult to track down errors that occur in the live production environment since the staging environment should have been the final validation. Not to mention the stress on engineers having to fight a live fire (think KCG). That’s where Princess comes in. Princess is the ideal staging environment that “uses production data stores, production network, and production hardware” (4). It was dubbed Princess in order to make a clear mental break from the old ideology of a “Staging environment” that’s similar to a production environment but not exactly.

In addition to providing a real production environment, Princess also is unaffected by any commits happening as your application is deploying. Instead those commits get queued up and can be deployed after the current deployment is finished (usually takes 1-5 minutes). A pain point with Jenkins and Ansible are that the whole team needs to stop development and take the time to deploy the application. If a commit is made while an application is deploying it will try to bunch that commit in with the rest of the stuff, and that could lead to unintended edge cases being deployed to production. Though it’s mostly automated, it wastes valuable time that engineers could spend further developing/designing the application.

In terms of architecture, Deployinator is relatively simple which makes it easy to use, integrate, and improve. At Etsy their whole stack consists of Chef for continuous development, Jenkins for QA/continuous integration, and Deployinator for continuous deployment. Due to the architecture of their stack, Etsy's able to deploy update to their applications over 50 times a day.

Today we have tools like Jenkins, Ansible, and Deployinator to help automate deployment. In the era where Knight Capital Group was founded, CI, CD, and CDL tools did not exist. Though at the time of their catastrophic deployment failure CI, CD, and CDL tools had started becoming an industry standard. The software models used to build these automated systems are all iterative and Agile in nature. Currently, many companies still are using some form of manual or semi-automated deployment. I believe that Etsy custom tech stack is a great example of the direction a successful automated deployment pipeline should strive towards. However, there needs to be a shift in the way the industry perceives automating software deployment.

For companies that are still manually deploying their software applications I'd propose an addition to all software systems building tools that are being used. All engineering teams use an integrated development environment (IDE). I would propose adding deployment tools native to IDE's so that engineers don't have to develop the software, and then develop a strategy to build a deployment pipeline to deploy it. In addition, all engineering teams use a software development kit (SDK) which varies depending on the type of system being built. If automated deployment tools are not combined into IDE's, for whatever proprietary reasons, they should at the very least be included in SDK's. Decoupling deployment from where engineers spend most of their time (their IDE and SDK) introduces many unnecessary complexities to the deployment phase of the software development life cycle. Companies in charge of developing IDE's and SDK's should work toward implementing native deployment tools such that engineers can do everything from one centralized location. The ideal solution would be to take Etsy’s custom deployment stack, or something similar, and integrate it to all the IDE’s and SDK’s being used today. This would require a big shift in the industry but would overall improve many of the pain points engineers face when deploying software systems.

Works Cited

Etmajer, M. (2016, December 06). Continuous Delivery 101: Automated Deployments. Retrieved April 29, 2017, from https://www.dynatrace.com/blog/continuous-delivery-101-automated-deployments/ **(3)**

Golub, S. (2012, November 29). Continuous Delivery vs Continuous Deployment vs Continuous Integration: Key Definitions. Retrieved April 29, 2017, from https://blog.assembla.com/assemblablog/tabid/12618/bid/92411/continuous-delivery-vs-continuous-deployment-vs-continuous-integration-wait-huh.aspx **(2)**

Kastner, E. (2010, May 20). Quantum of Deployment. Retrieved March 05, 2017, from https://codeascraft.com/2010/05/20/quantum-of-deployment/ **(4)**

Knightmare: A DevOps Cautionary Tale. (2014, April 17). Retrieved March 05, 2017, from https://dougseven.com/2014/04/17/knightmare-a-devops-cautionary-tale/ **(1)**