BASE2
SERVICES

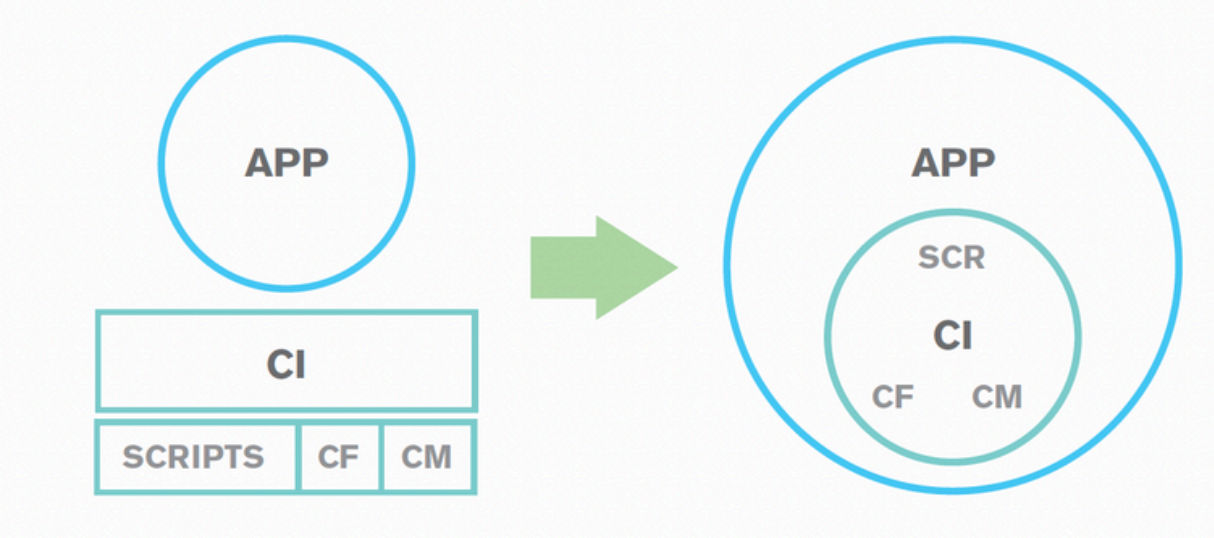
eBOOK

Start Building CI/CD as Code The 7 Lessons Learnt from Deploying and Managing 100s of CI Environments

Looking at Continuous Integration (CI) a bit differently

Traditionally, we have an Application where we would be building out code and then CI would be this thing we add on. It's this thing that sits out on the side, runs some tests and may deploy our application. Usually one person looks after the CI infrastructure and its really not part of the core of the application. It really is an afterthought.

Over the last 11 years that we have been building CI for ourselves and other organisations, we have learnt the importance of CI and the CI process and how it can really help with application delivery. A key learning over this time is the need to treat CI as a first class citizen in terms of the application and automation.



In this white paper we summarise the 7 lessons we have learnt from deploying and managing 100s of CI environments and how your CI/CD can be automated to create consistency and repeatability across environments with less reliance on individual team members, confidence in releases and less issues in production.

This stuff is hard
One of the things we have really discovered on our journey is that this stuff is hard. Building CI, scaling it across multiple teams and across an organisation - you will face a lot of challenges.

CI can be brittle: The pipeline can be broken easily when someone goes in and changes a job and does not really understand the implications of that change. Often enough, the pipeline breaks for someone else, someone who does not know a change has occurred and it becomes really hard to diagnose.

CI becomes the bottleneck: Before using CI, you deployed applications from people's laptops. It was fluid. When you moved it to a CI process it became fixed and rigid, which is a good thing for repeatability and removed the reliance on particular people's setups. Once you start to automate application deployments and builds, it often makes the deployment process more repeatable, but the bottlenecks start to happen when you have competing changes. You have not broken the CI process up enough to make change more granular. This quickly becomes the point of contention and really slows down the deployment process.

It gets hard to change: Changing the pipeline can have a ripple effect. You often hear "You can't change the pipeline because we have to be able to make deployments today". When a developer installs a new plugin, an upgrade to a plugin or changes the pipeline to satisfy what they need, they might break the pipeline for someone else.

Snowflakes: As developers start to build CI, they may install it on a spare PC on their desks, or in this day and age, in the Cloud, as it is very easy to get started. Very quickly after this, they start installing plugins to serve a purpose. Eventually, after several months, they don't really know what they have. It becomes a Snowflake that no one knows how to manage. If it becomes corrupted or fails, you need to rebuild it without knowing what state it was in. The CI environment is most likely not even backed up.

Duplication of effort: If I create a job in Jenkins that provisions an AMI using Packer to build a machine image, then someone will come along who wants to use the same job. They copy it, change the script on that job to work for them and now we have two versions of that script that are similar. They find a bug in it, make the change to one of the scripts to work for them, but do not change it for the other script. You are now left to manage two scripts that have diverged. If you want to make an across-the-board change, for example change the way you build AMIs, you have to do that in multiple places.

CI/CD is much more than just tools: Another perception is that buying a tool means I am doing CI/CD. It really is more than just using the tools. The tools are just a way to automate the process. What is important is the process itself and how the stakeholders interact with the CI/CD process.

The 7 lessons learnt
We have faced all of these challenges with many of our customers and have addressed them by building CI/CD as code. We tried to distil our accumulated knowledge into 7 lessons that can be incorporated into your current CI/CD capabilities.

Lesson 1: Treat CI/CD as part of your application

Treat CI/CD the same as your application
If your CI pipeline or environment breaks, it is just as problematic as a bug in your code in production. It has a major impact to delivering changes needed. So why would you treat it differently to the way you treat your application code?

Automate your CI environment like you would your application
You already build automation for your applications, automation for your applications' infrastructure and even spin up whole new environments using CloudFormation. Why not do the same for your CI environments? You can spin up a whole new CI environment as you would your application. Use the same tools as you would use for building your application. This way the knowledge is shared amongst your team. They already know how to build using CloudFormation and use tools like Chef for provisioning the app or deploying the app. Use the same tools to build CI because you already have the skills across your team to manage the CI environment.

Lesson 2: Continuously integrate your automation

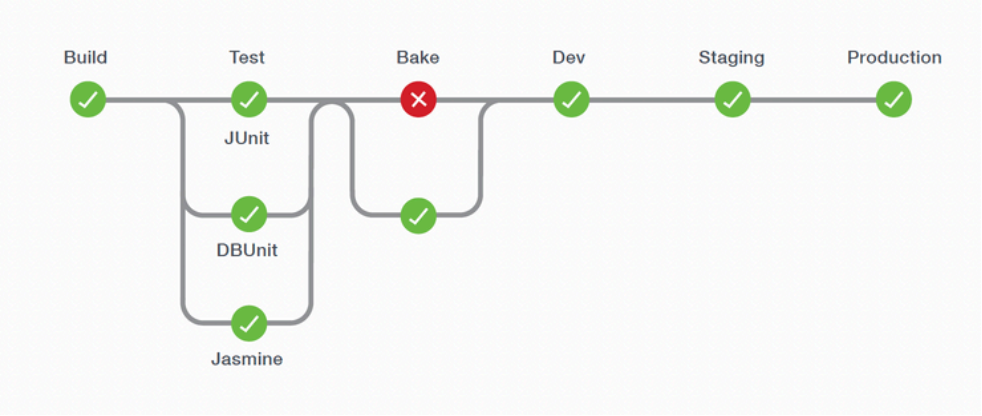
You build automation and it just works. It works well for a while. You go off to do something else, it does not get any love and eventually it breaks. If you are not constantly testing, changing and deploying your automation code, it starts to become stale and people forget what they need to do to fix it. The effort you put into the initial automation should not be lost.

Continuously integrate the application along with the infrastructure automation
You already automate the application, integrate, commit code, run tests and even more so now, automate infrastructure code. You also build CI processes too. You need to bring the application and infrastructure automation together as early as possible. Changes to either application or infrastructure code should be constantly tested together as early as possible, in the pipeline.

Foster ownership amongst developers and the operations team
Continuously integrating your automation helps establish a communication platform between the Devs and Ops. CI infrastructure is more than just the tools. It is the whole ecosystem, which includes source control management and how source control integrates with the CI tools. The automation and creation of a platform where you can collaborate, becomes the central place where developers and ops have a common language, common discussions and bring up the right ways to do things.

Make sure that you factor in the goal of constantly integrating your application and infrastructure automation so that you can fix issues early.

Lesson 3: Failure does not block progress



Failures are normal in a build pipeline

In a CI environment, creating a responsive, adaptable attitude to failure will help you advance. This is no different to code in your application. A change sometimes breaks code somewhere else and you need to fix it. You use unit tests to find breaks early in your application code, so you should be testing changes in your CI pipelines as well.

Make it easy to switch to the last working version

Build your pipeline to use the latest version. If the latest doesn't work, you should be able to quickly go back to one that does. At some point in time, something will break the deployment. For example, if as part of your CI pipeline you are building an AMI with the latest packages, and an updated package breaks the app, you should have the ability to go back to the last known working package or AMI. Use the latest versions of updates in your pipeline to avoid patching in production and keep the issues with patching at the development stage. If the latest version breaks, go back to the one that works until you have fixed the application to work with the latest version.

Use branching strategies

Use branching strategies to support deploying and testing combinations of infrastructure and application code. Deal with failures early by using a branching strategy that lets you deploy a variation of things and test those variations in development. Development teams can be working on new features as well as changing infrastructure code in parallel without blocking the main pipeline. Once those changes are tested in branches and accepted, they can be pushed through the normal pipeline.

Use branches to test CI pipeline changes

You should be able to change your build pipeline and not affect the main pipeline. You can test it in a branch, deploy it and see how it behaves. Then merge it in knowing that you have not broken anything.

Lesson 4: Make builds and deployments consistent for all environments

Traditionally, you may have the whole dev environment deployed on a single VM. Test might have the database split out and staging looks almost the same as production but not quite the same. You always end up with:

- "it works in dev",
- "doesn't work in test",
- or "worked in test but didn't work in staging or production".

Make deployments and environments the same
This is really the power of the Cloud. You should be able to make all your environments the same because you use the infrastructure code to provision all environments the same way each time. Once you have the automation in place, you can always have a production like environment in dev, test and staging. You can have varying scales of production with smaller instances, less instances in auto scale groups, etc. However, in terms of architecture and network topology they are always the same.

The deployment process should be the same for all environments; development, test and production
However, this may not be fast enough for feedback in development, especially if you are using AMI promotion. You may want to use AWS CodeDeploy for every single commit into development, and when the developer is happy, have a process that builds an AMI. You also want to make sure that you deploy this AMI into development first before you deploy to other environments. This way, you are truly ensuring your application code works with the infrastructure code across all environments.

Prefer immutable artifact promotion to in-place deployments
As explained above, you don't have to run the entire process each time you make code changes. You just want to ensure that before you release code for test, the full process is run, creating immutable artifacts (e.g. AMIs) that are then promoted through each environment. This ensures a higher guarantee of repeatability.

Lesson 5: Just commit code
Typically, you build out a CI pipeline and you may have automated some of this. But what you will find is that you will be logging into the CI server, executing a job, and maybe needing to execute another and another in manual execution steps. You have to remember various parameters for a job, occasionally putting in the wrong parameters and deploying to the wrong environment or pushing out the wrong build. This gets very messy very quickly and sometimes you need to remember the many steps to deploy your code.

Focus on creating a Developer Experience
Create an experience for developers using the tools that they are really comfortable working with. Often this is source control. You want to limit the number of interactions that the developers have to make with the CI tool. If a developer does not have to log in to the CI tool and is just committing code, you have significantly improved their experience with the tools. Trigger executions automatically from source commits and you will find developers will embrace the process.

Create a platform where developers can interact where it makes sense
Developers should be interacting in the code. An example of this is: raise a pull request, add a comment to the pull request and merge the pull request, which triggers a deployment build to an environment. Use pull requests to trigger a process to promote code to an environment. This lets the CI tool do what it is supposed to do without having to log in and run those jobs, allowing the developers to interact in the tools they are most comfortable with.
A good example of this in practice is:
1. A developer commits the code
2. CI tool runs the unit tests and gives it a thumbs up
3. A reviewer then comments “ok for dev”
4. CI tool runs the build process and deploys to dev
5. CI tool then runs the automated tests and marks it “pass/fail”
6. If you are doing CD, then the CI tool would push to staging, test and then to prod

Fast feedback
Fast feedback of job failures and the promotion of changes should provide notification in places that make sense. Which usually means communication tools used by the team (Slack, HipChat or IRC) and a comment in the pull request that executed the job. Always add comments to pull requests with the status of the build at each step of the pipeline. This way all you have to do is provide the proof of the execution and its status to a manual change approval process within the organisation (i.e. the Change Approval Board (CAB)).

Refactor the build pipeline often
Typically the pipeline is built to meet the minimum requirements for building and deploying applications, e.g. being able to deploy from a single branch into development. Later these requirements change, e.g. developers may also want to deploy from arbitrary feature branches into development. So you need to refactor the pipeline. If you have the right branching strategies and automation in place, refactoring is not a big deal and can be done as often as required.

Lesson 6: Expose build artifact details

```
curl https://apply.base2.services/health | jq '.'

{
 "status": "OK",
 "version": "2.3.193",
 "build": "193",
 "build_date": "2017-04-01",
 "git_sha": "0e46f44da4277ce7e6da910ad429e1d3c4a325b5",
 "cf_version": "231",
 "chef_version": "1.201",
 "db_version": "42"
}
```

Typically, people become very attached to the build history and the information that is captured in the CI tool. Being able to look at the history of the build jobs is useful but you should not be reliant on accessing the CI tool to get information about what version was deployed where.

Metadata should be encoded in the artifacts
As the artifacts travel through the pipeline, more and more metadata is encoded in the artifacts. For example, information is encoded in the CloudFormation templates about what version of the application is referenced in an AMI. This makes each artifact self-describable and you can clearly and confidently say that this set of artifacts deployed this version of the application in this version of the environment.

Ensure that the metadata details are discoverable
This may be achieved by building a health check end-point that also returns this metadata. You can then put together a dashboard about what artifacts, versions and parameters are in each environment. If something is not working in an environment, you can get more information for debugging as needed when details are query-able. Differences in versions can be tested, and if a version of an artifact is different to what you expected, you can ask why.
You no longer have to rely on the build history of your CI tool to find this information.

Lesson 7: Everything as Code

```
# VPC Stack
Resource('VPCStack') {
  Type 'AWS::CloudFormation::Stack'
  Property('TemplateURL', "https://#{source_bucket}.s3.amazonaws.com/ciinabox/#{ciinabox_version}/vpc.
json")
  Property('TimeoutInMinutes', 5)
}

# ECS Cluster Stack
Resource('ECSStack') {
  Type 'AWS::CloudFormation::Stack'
  Property('TemplateURL', "https://#{source_bucket}.s3.amazonaws.com/ciinabox/#{ciinabox_version}/
ecs-cluster.json")
  Property('TimeoutInMinutes', 5)
  Property('Parameters',{
    ECSCluster: Ref(cluster_name),
    VPC: FnGetAtt('VPCStack', 'Outputs.VPCId'),
    RouteTablePrivateA: FnGetAtt('VPCStack', 'Outputs.RouteTablePrivateA'),
    RouteTablePrivateB: FnGetAtt('VPCStack', 'Outputs.RouteTablePrivateB'),
    SubnetPublicA: FnGetAtt('VPCStack', 'Outputs.SubnetPublicA'),
    SubnetPublicB: FnGetAtt('VPCStack', 'Outputs.SubnetPublicB'),
    SecurityGroupBackplane: FnGetAtt('VPCStack', 'Outputs.SecurityGroupBackplane')
  })
}
```

```
# ciinabox services configuration

services:
 - jenkins:
     ContainerImage: base2/ciinabox-jenkins:2
 - openvpn:
     config: s3://vpn.base2.services/ciinabox-vpn
     users:
       - aaron.walker
       - leeroy.jenkins
```

```
include_recipe "base2-chef-handler-notifier"

template "docker_images_to_slack.rb" do
 source "docker_images_to_slack.rb.erb"
end

case node['platform_family']
when 'debian'
 include_recipe 'apt'
when 'rhel'
```

```
defaults:
 scripts_dir: ciinaboxes
 github:
   credentials: github
   org_white_list:
     - base2Services
   cron: "* * * * *"
 slack_channel: "#ciinabox"

jobs:
 -
   name: base2-cookbook-pull-request
   repo: base2Services/base2-cookbook
   folder: TheFactory
   shell:
     - file: scripts/chef-build.sh
 -
   name: base2-cookbook-release-deploy
   repo: base2Services/base2-cookbook
   branch: master
   folder: TheFactory
   parameters:
     cookbook: base2
   shell:
     - file: base2/scripts/chef-release-deploy.sh
   push:
     - develop
     - master (edited)
```

If you have a CI pipeline make sure it is re-creatable. Developers should be able to delete the CI environment one day, then recreate it the next day, and it should be in exactly the same state it was. Leverage the same tools you use for creating and configuring your application and environments for creating and configuring your CI environment.

It is important to understand the benefits of having your CI environment as code. It is easily replicated across multiple teams. The amount of reuse for those teams will go up because you have created a shared repository of standardised build jobs and you are not dependent on the person that configured Jenkins. Also, anyone within the team should be able to make changes to a pipeline or the CI environment because you are using the same tools.

It is all managed in source control. All changes are trackable and you can easily revert if a build script is broken. You are creating high visibility of your pipeline just like you would normally with your application code.

.

So what now?

The goal that you really are trying to achieve is that CI is not an afterthought. It should be part of what you do and not this separate thing. But you should not have to think about it. CI becomes a capability that is created and a team should be able to easily onboard that process to other team members.
Developers can then focus on what the build pipeline is and the steps they need to do rather then worry about how they are going to do it. It should just be provided. All the management of the CI tool really just becomes the same as the application, with the pipelines evolving as the application evolves.

Benefits of this approach

Implementing this approach and facing the challenges we have had over the past 11 years has had many benefits for our company. We are able to scale easily with new customers coming on board every month and our team is constantly improving and reducing the effort required on the core of CI/CD. Some of the key benefits our customers have had are as follows:

- Go faster, get ahead, stay ahead and improve customer satisfaction by focusing on application features. Production builds are more consistent and can happen more often.
- Faster reaction times to issues. The environments are exactly the same, developers are comfortable with the tools and are able to make change faster.
- More confidence, lower risk and higher visibility. Constantly making change and testing change often in your CI pipeline increases the confidence in deployments at production release time.
- Better team communication with trust and visibility. Better communication means the whole organisation gets improvements. Teams are able to work faster and provide accurate feedback on the status of builds
- Scale, you are building repeatability. The onboarding of new projects and new staff is easier when you are focused on what needs to be done rather than how to do it.

So just start Building CI/CD as Code.

Get there faster

We are happy to help you discover how you can deliver results in the cloud by using CI/CD as code. base2Services provides DevOps as a Service to development teams and ambitious organisations to help them build and manage innovative, scalable and agile cloud based solutions. With more than 10 years of experience working with DevOps practices and Amazon Web Services (AWS), we help our customers transition to autoscaling, self-healing environments on AWS and give them the ability to deploy as often as they need integrating CI/CD tools and repeatable processes.

BASE2 has a strong and longstanding partnership with AWS and many firsts in the market. It was Australia's first company to launch a managed AWS offering, was named Australia's first AWS Advanced Consulting Partner and Australia's first AWS Managed Service Provider Competency Company. Aaron Walker is an AWS Certified APN Cloud Warrior and our staff hold several AWS certifications including AWS Certified DevOps Engineer – Professional, Solution Architects (Associate and Professional), Developers and SysOps Administrators.

Get in touch

BASE2 Services is a global leader in Cloud Delivery, Operations and Management. Specialising in DevOps, AWS and cloud-native computing.

We enable organisations to accelerate innovation through automated, highly secure, repeatable and scalable cloud-based solutions. Contact us today to find out how we can help you.

USA

11801 Domain Blvd
3rd Floor
Austin TX 78758
+1 646 586 9485

AUSTRALIA

Level 3
44 Clifton St
Prahran VIC 3181
1300 713 559

GERMANY

4. OG
Potsdamer Str. 182
10783 Berlin
+49 30 2000 5370

info@base2services.com

base2services.com.au