CI/CD for Infrastructure & Configuration Management

with Otter + BuildMaster

inedo

Abstract & Overview

Organizations of all sizes, from the leanest startup to the stodgiest enterprise, use CI/CD practices to greatly improve production and delivery of software. This yields higher-quality software at a lower cost and allows businesses to deliver ideas to market faster.

As development teams adopt CI/CD practices, they start delivering new applications and releases faster and faster. This constantly changing software inevitably requires changes to infrastructure and configuration, but many operations teams aren't accustomed to the pace—nor do they have the proper tools required.

Trying to directly apply successful CI/CD practices for applications is highly unlikely to yield success for infrastructure and configuration changes. In this guide, we explore:

- How and why CI/CD has been so successful for application changes
- Infrastructure and configuration changes as a new bottleneck
- Challenges with CI/CD for infrastructure and configuration changes
- How to overcome challenges and implement CI/CD for infrastructure

We include a hands-on guide for how to implement this with BuildMaster and Otter.

You can do everything in this guide with BuildMaster Free and Otter Free editions!

About Inedo, Otter, and BuildMaster

We help organizations make the most of their Windows technology and infrastructure through our Windows-native and cross-platform DevOps tools.

- BuildMaster, a tool designed to implement CI/CD, automates application releases.
- Otter manages infrastructure.

Harnessing the power of both tools allows users to manage infrastructure while enjoying all the benefits of CI/CD.

Contents

CI/CD for Applications: A Quick Refresher

Continuous Integration (CI) and Continuous Delivery (CD) refer to a set of practices development teams can use to produce and deliver software for business teams. CI/CD practices are not uniformly defined by a standards body such as ISO, but instead are explored, implemented, integrated, and continuously refined as part of an organization's core business processes.

Although specific CI/CD processes evolve differently within each organization, all organizations tend to experience the same benefits:

- Better quality software with fewer defects through production
- Faster delivery of business ideas to market by enabling faster release cycles and allowing changes in days or weeks rather than months or quarters
- Cheaper implementation across the entire lifecycle, including less time spent coding, deploying, and testing software changes

These seem almost too good to be true, yet organizations in all domains, from banking to games, reap benefits.

While this guide doesn't dive into the details of how to implement CI/CD for applications, we cover the key mechanisms that drive success (pipelines) and demonstrate how you can apply CI/CD practices to infrastructure and change management.

Pipelines: The Heart of CI/CD

One of the key components of Continuous Delivery (CD) is the deployment pipeline. Pipelines model the software release process by defining the servers and environments that builds will be deployed to as well as the manual and automatic approvals required at each stage of the process. Different applications may use different pipelines, or the same application may use different pipelines for different releases.

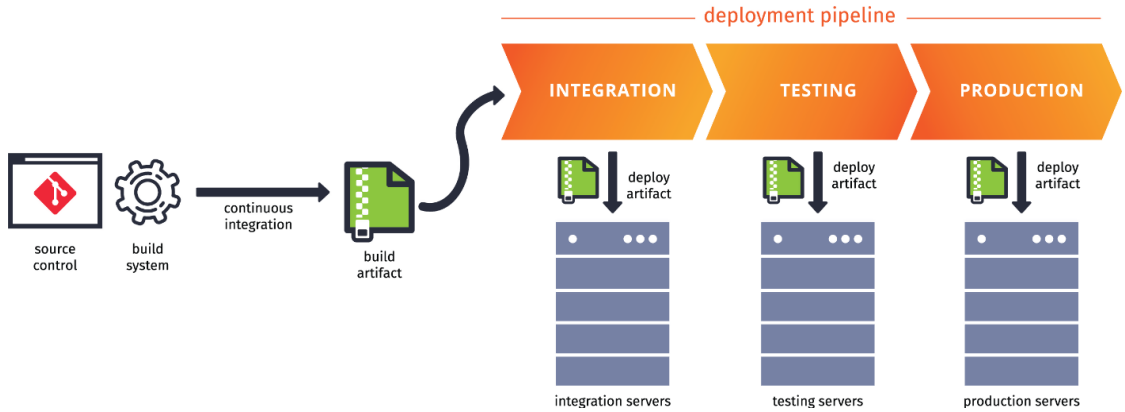
For example, a basic web application might use a pipeline with only two stages (testing and production) and simply deploy to a different folder on the same server. Another application may require a dozen stages, each with multiple targets that go to different

environments and with all sorts of automatic and human approvals to meet compliance requirements.

End-to-End Software Delivery with CI/CD

Deployment pipelines require an input (i.e. the build to deploy), and that input comes from Continuous Integration (CI). Specifically, this input is a build artifact: essentially a zipped file containing the code that will be deployed to each server across each environment.

When you combine CI's build artifacts and CD's deployment pipelines together, you have the essence of CI/CD: end-to-end software delivery.



The Cardinal Rules of CI/CD

Although the implementation details of a CI/CD process differ from organization to organization, they will all share a set of cardinal rules:

- One codebase, many environments.
    - DO build one version of your application that's used by all of your environments from the same codebase or branch in source control.
    - DON'T build an "integration version" of the application from an integration-quality branch then test that version and merge changes to a testing branch. Likewise, don't build a "testing version" for the testing environment. Don't test and merge all the way to production.

- Deploy the entire application or component.
    - DO deploy the entire build artifact produced by the CI process to each environment. Let your deployment tool optimize this process such that unchanged files on disk are not actually copied over a network.
    - DON'T cherry-pick files that changed between each environment to mitigate risk.
- Model process with automatic and manual gates.
    - DO strike a balance between automated and user testing modeling business processes and bringing development and business teams together.
    - DON'T force a fully-automated process or fully-manual process on business users.
- Ensure traceability and auditability at every step.
    - DO ensure that every production deployment can be traced back to a change in source control and can be audited to see who approved it and when.
    - DON'T bypass the process through emergencies; instead, build an emergency process.

A CI/CD tool like BuildMaster (inedo.com/buildmaster) facilitates and implements these practices.

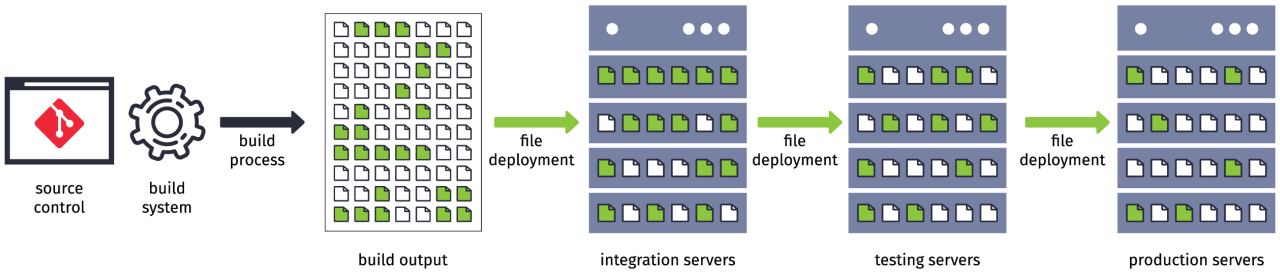
Software Delivery without CI/CD

Without an end-to-end software delivery process, releases are often deployed by a release engineer on an ad-hoc basis at the direction of someone on a business or development team. Although these deployments are usually not manual (i.e. by logging into servers and copying files), release engineers use a deployment tool or script that models a manual process.

Manual delivery processes mitigate risk by deploying infrequently (since fewer deployments means fewer production failures) and by cherry-picking files. This starts by

having developers use a build process (either CI or manual) and specifying which files from a build output to deploy to the first pre-production environment.

The application is tested in the first environment. The developers then specify which files to deploy to the next environment based on which parts of the application passed testing. The process continues to production.



At each step in the process, a different set of files may be deployed to each server, and different sets of files comprise the applications in each environment. Ultimately, a different application is being tested in each environment leading to:

- Lower quality software in production because testing at all levels was too difficult
- Slower delivery of changes to production from manual and ad-hoc processes
- Costly implementation from bug fixes and arduous testing/deployment processes

CI/CD Implementation Challenges

Although CI/CD can greatly improve software delivery processes, implementation often challenges technically. The largest and most difficult-to-address challenges tend to be cultural and behavioral.

Everyone involved in the delivery process—from release engineers to testers and business stakeholders—is accustomed to the way things are. They recognize it's suboptimal, but changing it could make things even worse and, ultimately, make their lives more difficult.

Like all organizational changes, it's important to get "buy-in" from those committed to the existing process. That often involves "selling" them on the new process. With modern

CI/CD tools like BuildMaster, you can accomplish this with a quick demo and proof of concept.

Without CI/CD tools allowing you to easily prove the value of changing processes and how easy is to adopt, getting others to buy in can be nearly impossible.

The Configuration Management Bottleneck

As development teams adopt CI/CD practices, they deliver new applications and releases faster and faster. This constantly changing software inevitably requires changes to infrastructure and configuration, but many operations teams aren't accustomed to the pace—nor do they have the proper tools required.

Defining Configuration

"Configuration" is anything you do to a server (i.e. your infrastructure) after a fresh install of the operating system. On Windows, this includes things like:

- Adding Windows Features such as IIS
- Creating an Application Pool
- Opening a port on the Windows Firewall
- Installing a Windows Service

Most server configuration will not have an impact on the software and applications running on the server. For example, adding IIS will have no impact on an installed Windows Service.

Some configurations directly control how your applications run. For example, a site's configuration in IIS determines which port the hosted application uses, and the Application Pool's configuration determines the running identity or Windows account.

Other times, configuration will indirectly impact your applications, sometimes in undesirable ways. For example, if you enable Windows Firewall and block the listening port for your application, then users will not be able to access the application.

Managing Infrastructure & Configuration Changes

Making an actual configuration change is easy, particularly on Windows. Just perform the necessary steps in the UI, or run a PowerShell script. However, managing all of these configuration changes—whether they've already been made or need to be made—can challenge.

Configuration management is important to rebuild the server in the event of an upgrade or hardware failure. However, it's also good for auditing purposes by letting others see what changed for compliance reasons or to diagnose why something "stopped working."

When you have testing servers used to validate application changes before production, configuration management is critical. For example, if an Application Pool on your production server runs under NETWORK SERVICE, but the testing server's Application Pool uses LOCAL SYSTEM, this could lead to difficult production bugs and outages—the exact thing the testing process should mitigate.

Thus, there are three important aspects to configuration management:

- Intended state. In a Word document, a configuration management database, or an "infrastructure as code" file, documenting the intended state of a server lets anyone see the server's configuration without having to log in to that server
- Change execution. Manually, with a script, or via a change management tool, how the configuration change will actually be performed against the server
- Change tracking. With a ticketing system or email, keeping track of proposed and actual changes helps auditors and developers

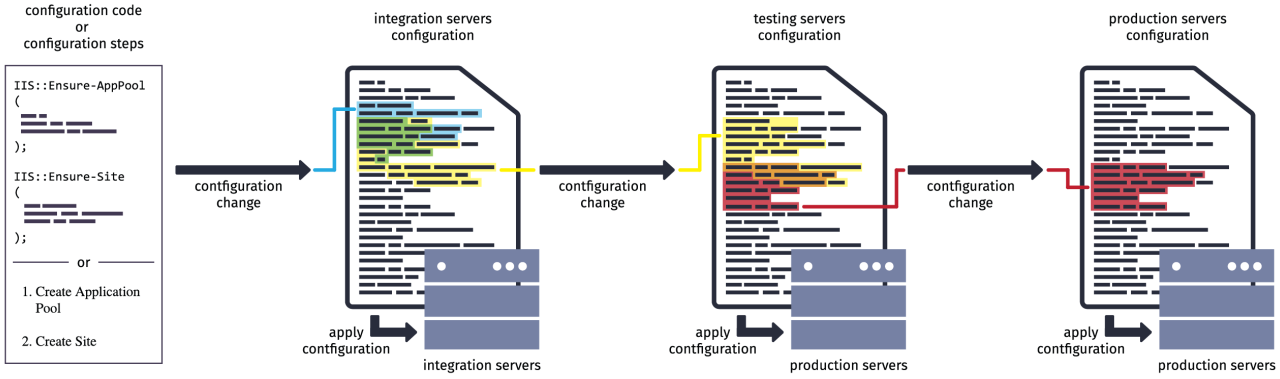
Development-driven Infrastructure & Configuration Changes

New applications and releases often require configuration changes. For example, a new web application needs both an Application Pool and a Site in IIS with specific settings.

An operations engineer typically makes configuration changes on an ad-hoc basis at the direction of a development team. Sometimes these changes are executed manually, but operations engineers also use a configuration management tool to automate execution.

Because changes are tested in pre-production environments, they need to be applied to all servers across all environments used by the application. This starts by having developers specify configuration changes (either in a script or a ticket) to apply to the first pre-production environment.

The changes are then applied and tested in the first environment. Developers then specify which configuration changes to apply to the next environment based on which configuration performed properly. The process continues through to production.



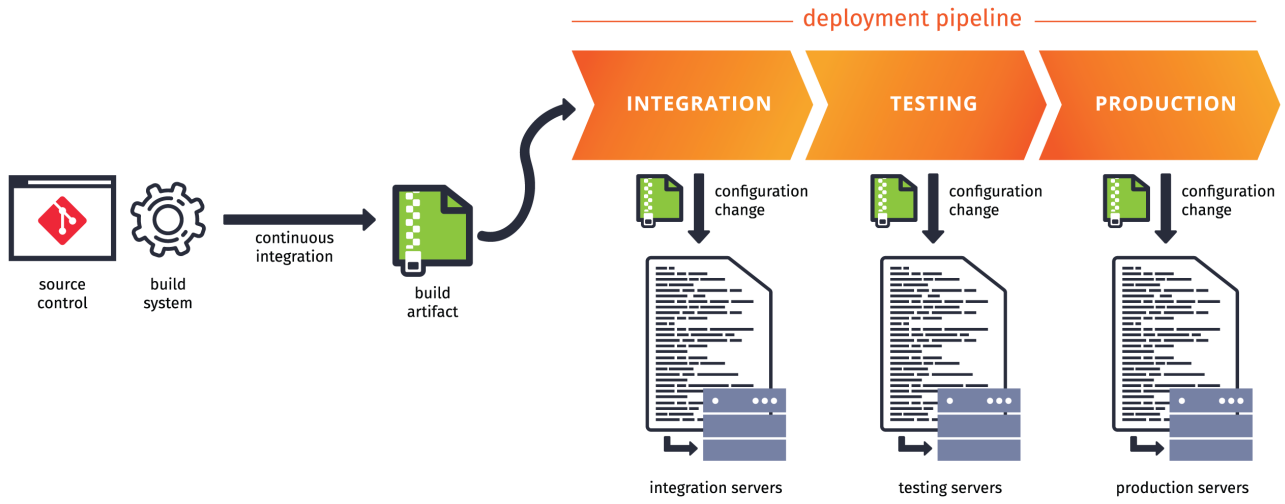
This ad-hoc, piecemeal process is very similar to a software delivery process without CI/CD and yields the same problems:

- Lower quality software in production because testing at all levels was too difficult
- Slower delivery of changes to production from manual and ad-hoc processes
- Costly implementation from bug fixes and arduous testing/deployment processes

However, as applications release more often and on shorter schedules, they require more development-driven configuration changes leading to a snowballing problem operations teams will never have the resources to manage.

Infrastructure & Configuration Deployment Pipelines

Fortunately, you can leverage deployment pipelines to deliver configuration changes. This introduces the same benefits development teams experience in applying CI/CD.



While the result looks very similar to CI/CD for applications, the means are very different and require an advanced CI/CD platform, like BuildMaster, and an advanced configuration management tool, like Otter. Both are easy to setup and use.

CI/CD and Infrastructure Configuration Generally Don't Mix

Trying to directly apply CI/CD practices used for applications is highly unlikely to yield success. There are many technical and cultural reasons for this.

Different Missions: Development vs. Operations

Two different teams with two different missions maintain applications and infrastructure.

- Development focuses on change. Business teams constantly explore new ways to change existing applications; while, development teams are constantly pushed to accelerate change cycles
- Operations focuses on stability. More than anything, business teams require all infrastructure and applications without downtime; while, operations teams avoid changes to minimize risk of failure

These teams use different automation tools and apply distinct change policies.

Different Auditing & Compliance

Despite divides, both teams share a common goal in change tracking.

For development teams, this is relatively easy—even with poor CI/CD tools and setups. Because source control serves as the "source of truth" for what the application code should be (even without proper traceability), you can use file dates and sizes to "backtrack" to the person who committed the code and hunt down the reason for change.

This approach, however, doesn't work for infrastructure and configuration changes. There is no built-in "change record" like source control, and the "source of truth" is the actual server itself. By only inspecting a server, there's no way of knowing when it was configured, who configured it, or why it was configured.

This is why operations teams tend to develop arduous change management processes and why development teams don't quite understand their importance.

Different Paradigm: Applications vs. Configuration

Applications are comprised of files on disk. Updating applications involves replacing files on disk. You can easily backup applications by zipping these files and rollback to a previous version by restoring the files.

Servers, on the other hand, are much more complicated. They are updated by performing a series of steps in the UI or with a script. You can't easily backup or restore a server. If you want to "rollback" to a previous configuration, you must apply a different set of steps.

While server configuration can be abstracted using a "declarative configuration" script (such as OtterScript), the automation tool processing the script performs the same steps you would otherwise run manually. Consider this simple OtterScript statement:

```
IIS::Ensure-AppPool AccountsWeb ( Pipeline: Integrated );
```

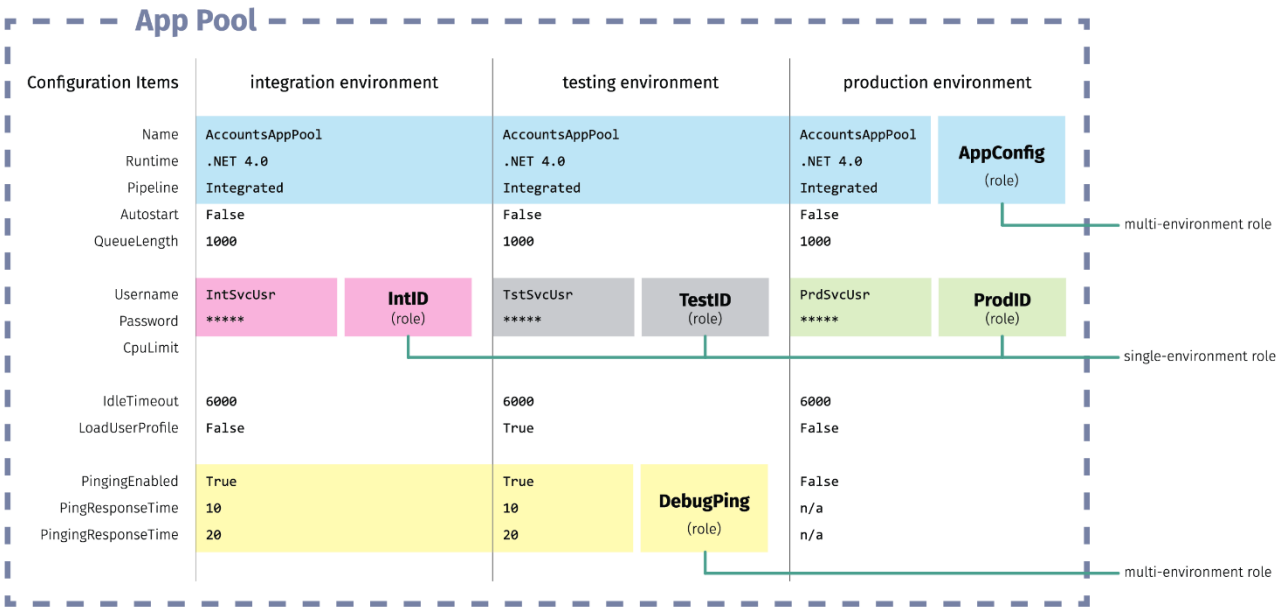
Upon seeing this statement, Otter checks for an Application Pool named AccountsWeb and creates it if not found. Otter then makes sure the Application Pool's "Pipeline" setting is configured as Integrated, and that's it. No other configuration changes are made to the AccountsWeb Application Pool or any other Application Pool unless explicitly defined.

Compare this to an application deployment. Whenever you deploy a file, the entire file is replaced rather than a “line item” within the file.

Different Configurations for Different Environments

Even if they’re running the exact same applications, servers are inherently different and often require different configurations. This is true for servers in different environments such as production and pre-production testing environments.

Some configuration sets (i.e. configuration roles) are identical across all environments such as the core requirements for a particular application. Other configurations are only used in pre-production environments (like debug settings) or are unique per environment (like identities):



Different configurations for different environments fundamentally clashes with the “one codebase, many environments” cardinal rule of CI/CD.

Different Types of Configuration Changes

A lot of infrastructure and configuration changes will occur outside of application release cycles such as operating system patches, security hardening, monitoring, and so on.

These types of changes impact all software running on the server, including third-party applications and services.

Because these configuration changes may impact server performance, they are tested in a similar manner to application changes prior to production. Operations teams want to use the same processes to test all types of configuration changes to ease fast-paced application release changes.

Adapting CI/CD Pipelines for Infrastructure

Although there are a lot of technical differences between applications (code) and infrastructure (configuration), one of the hardest things to overcome when adapting CI/CD is the mental model. Operations teams will need to be comfortable with "infrastructure as code," and development teams will need understand that "infrastructure as code" isn't actually code.

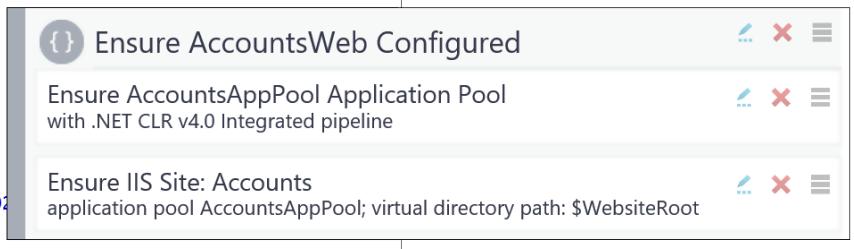
Declarative Configuration Management

Although there are many ways you can manage server configuration—from arduous, hand-crafted spreadsheets to heavy-handed configuration management databases—the concept of "Declarative Configuration Management" has evolved as the best way to keep track of constantly and rapidly changing configurations.

Declarative configuration management allows you to define the desired end state, not the steps required to attain it. This "Desired configuration" approach was pioneered over a decade ago with open-source tools for Linux such as Chef and Puppet. These tools allow infrastructure-savvy developers to paint a picture of a successfully running application for code-savvy operations engineers.

As a Windows-native DevOps tool, Otter brings this approach to Windows and makes it easy for both teams to understand and change configurations. Here's what a declarative configuration plan for a simple web application looks like using OtterScript:

```
4  # Ensure AccountsWeb Configured
5  {
6      IIS::Ensure-AppPool AccountsAppPool
7      (
8          Runtime: v4.0,
9          Pipeline: Integrated
10     );
11
12     IIS::Ensure-Site Accounts
13     (
14         AppPool: AccountsAppPool,
15         Path: $WebsiteRoot,
16         Bindings: %(IPAddress: 192
17     );
18 }
19
```



While the approach is similar to Linux-native tools, OtterScript lets you switch back-and-forth between visual and textual modes.

When this configuration plan is applied against a server, Otter *ensures* that the desired configuration exists, and either create the necessary Application Pool and Website or update it to match the specified configuration. Otter also tightly integrates with PowerShell DSC, allowing you to use DSC Resources to configure virtually everything on a Windows server.

The key benefit of declarative configuration is that you don't need to worry about the actual configuration on the server. Change the configuration plan, and the server's configuration will be changed to match the plan.

Server Roles

A server role is essentially an OtterScript configuration plan, and it's used to define a specific set of configurations that can be assigned to any number of servers. Roles are used to describe any type of configuration from application-driven settings to security hardening or compliance policies.

For example, you may have an `iis-server` role that ensures IIS is enabled as a Windows feature as well as an `accounts-web` role that ensures a specific web application is configured properly. The `iis-server` role would be applied to all web application servers, while `accounts-web` would only be applied to servers that run the AccountsWeb application.

Although servers in each environment will often have a different configuration, you should generally avoid creating environment-specific roles, especially for an application-

based configuration. This is because environments are generally used to test and validate changes before production. If you're not applying the exact same set of configurations (i.e. roles) to these environments, then you're not really testing the same things. This largely defeats the purpose of pre-production testing.

For example, if the AccountsWeb application is stored in different paths on production and testing servers, you should not create `accounts-web-test` and `accounts-web-prod` roles, but instead use a variable like `$WebsiteRoot`, and then configure that variable value on each server.

Infrastructure as Code

OtterScript defines role configuration plans and stores them as text files. These roles depend on other roles and use assets like OtterScript modules, PowerShell/shell scripts, and configuration files. All of these remain declarative in nature: when you edit a configuration role, the server's configuration updates to match the role's plan.

This technique is generally referred to as "infrastructure as code." While these files aren't traditional application code, you can store them in a source control (Git) repository for versioning and change-tracking purposes. This also allows your source control repository to be the "source of truth" for infrastructure and configuration changes.

Rafts & Related Configuration

Otter can bundle roles and related configuration assets into an abstract file system called a raft. For example, you could bundle all of the required configuration for a particular set of applications that are maintained by a single development team into a raft they have permission to maintain.

These rafts can be backed by a Git repository, meaning changes to raft files from within the Otter UI will automatically be committed to the Git repository. You can also commit changes to the Git repository outside of Otter, and Otter will then apply those changes to servers.

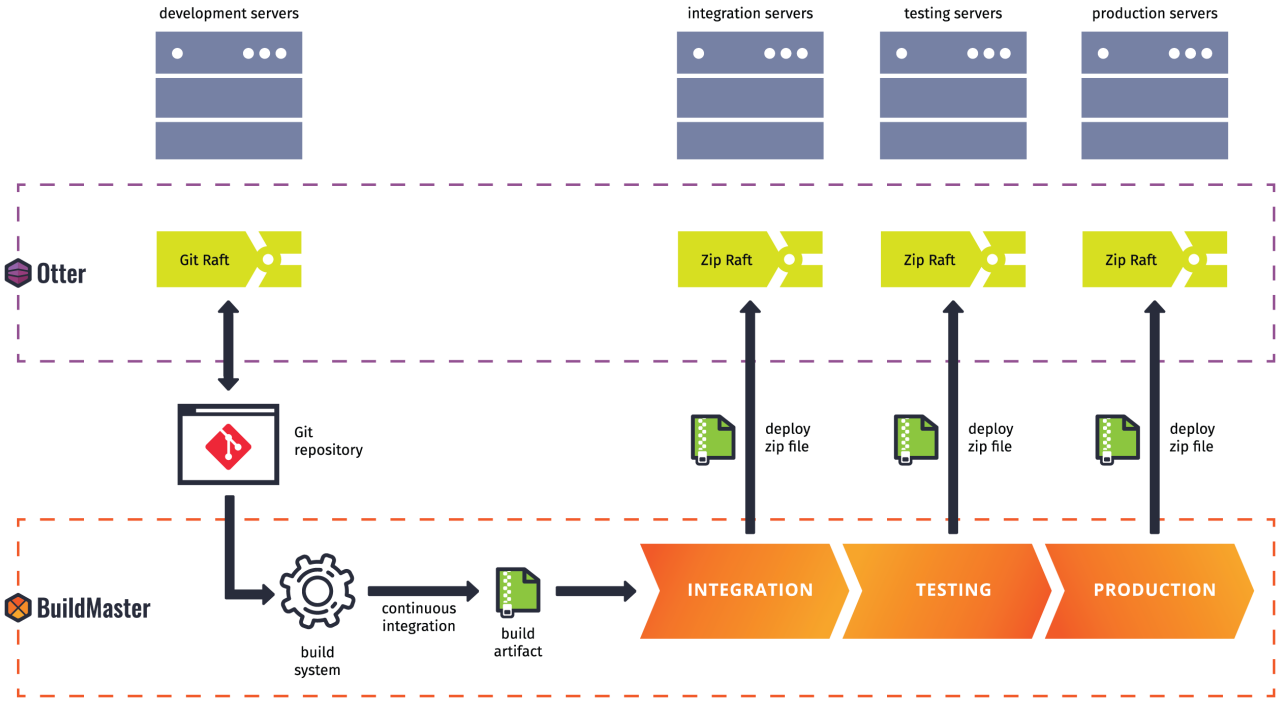
Rafts in Otter can also be backed by a zip file, meaning they would be read-only and un-editable from the Otter UI. While this can be seen as a hinderance, they allow you to

enforce the same rules you would apply to an application deployed to a server: no piecemeal editing of files.

CI/CD for Infrastructure

Once you have your infrastructure code in a Git repository, you can use a tool like BuildMaster to retrieve that code from that repository and then create a build artifact. BuildMaster can then use a pipeline to deploy build artifacts to the Otter server to use a zip-based raft.

Ultimately, the CI/CD for Infrastructure pipeline looks very close to the CI/CD pipeline for application.



Like CI/CD for applications, you should always use a pipeline to deploy changes. This means starting in source control and deploying those changes through all environments. If you need emergency changes to bypass testing processes, simply build an emergency pipeline.

CI/CD for Infrastructure & Configuration Management

Hands-on with Otter + BuildMaster

BuildMaster, a tool designed to implement CI/CD, automates releases for applications. Otter is a tool designed to manage infrastructure. Harnessing the power of both tools allows users to manage infrastructure with all the benefits of CI/CD.

You can do everything in this guide with BuildMaster Free and Otter Free editions!

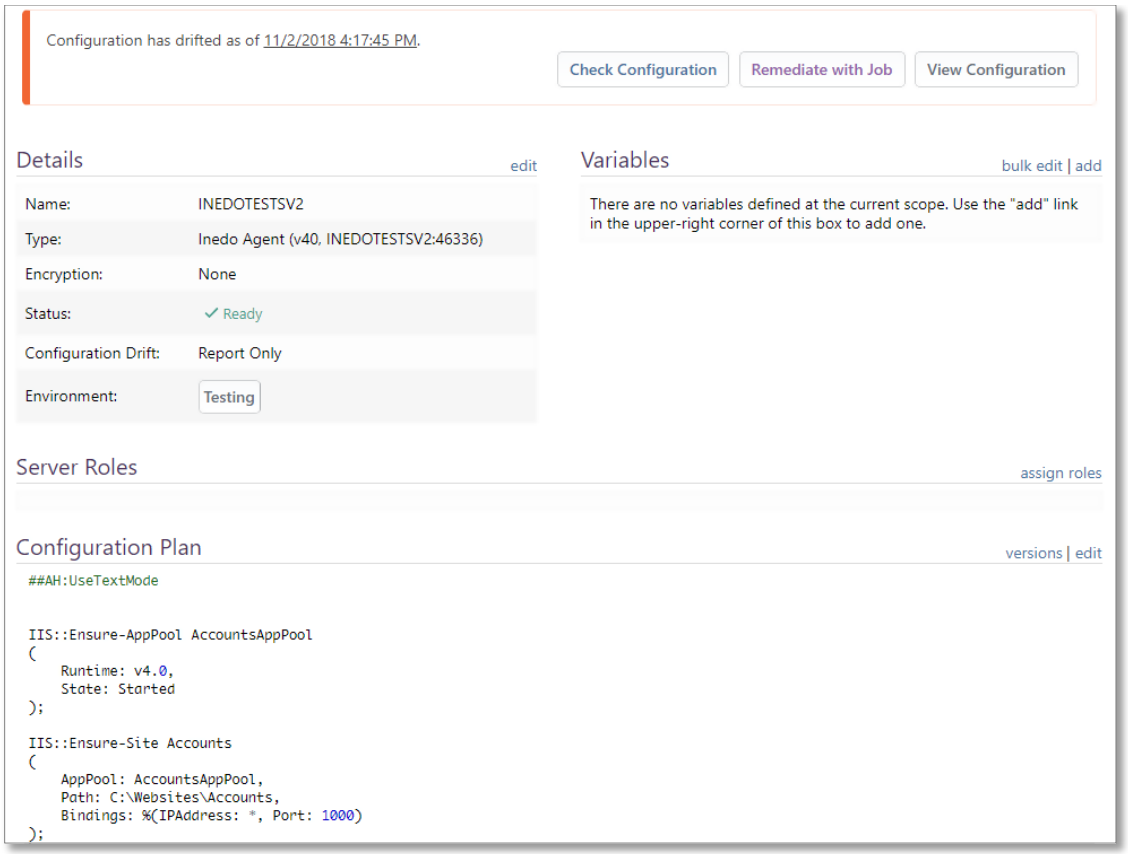
Contents

Basic Set-up: Versioned Configuration Management with Otter

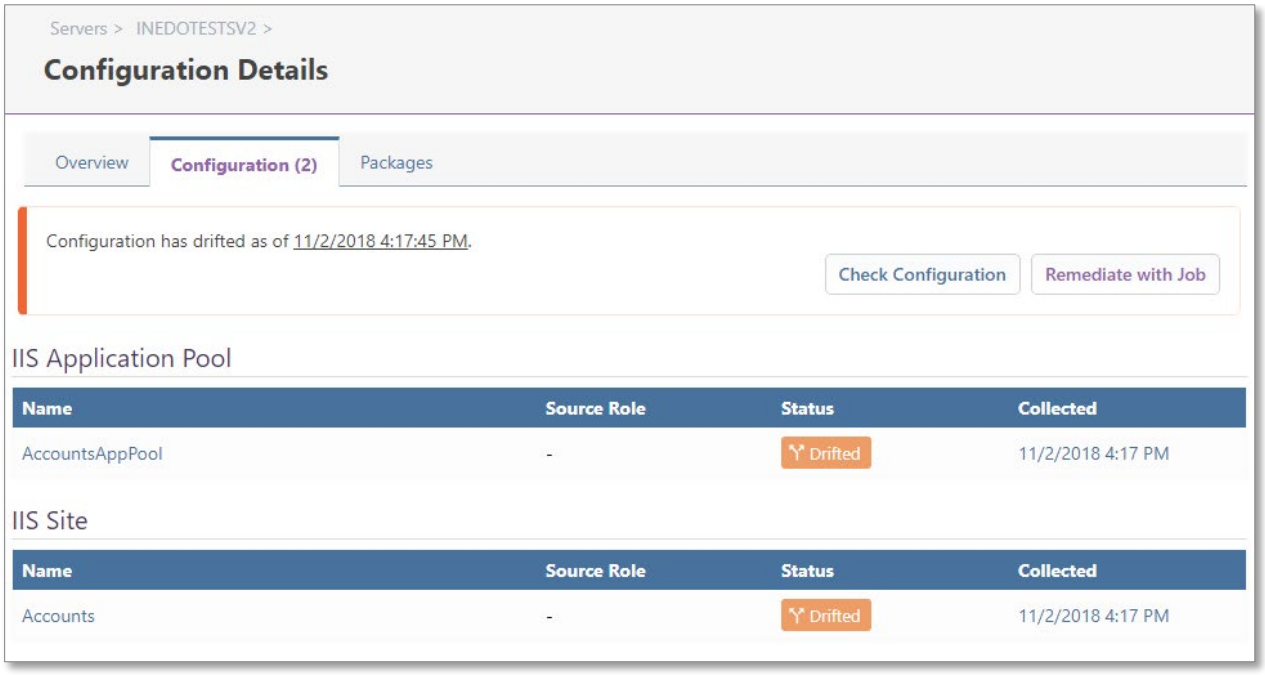
Otter manages full server infrastructure via the "Infrastructure as Code," or IaC, paradigm. Effectively, OtterScript is the DSL used to define infrastructure on a server directly or a server role (assigned to any number of servers). Any server or role in Otter can be configured to automatically remediate drift or to facilitate change tracking by monitoring drift only.

This infrastructure code and related resources (e.g. servers, variables, orchestration plans) are stored in a "raft," an extensible filesystem of sorts. By default, these resources are stored in the Otter database, but another common raft is the Git raft, transforming any changes to OtterScript into commits within Git. This opens the door to all the benefits provided by a full source control management system including complete version history, diff reports, ability to edit outside of Otter, and more.

In the following figure, we see a basic configuration of a server in a testing environment:



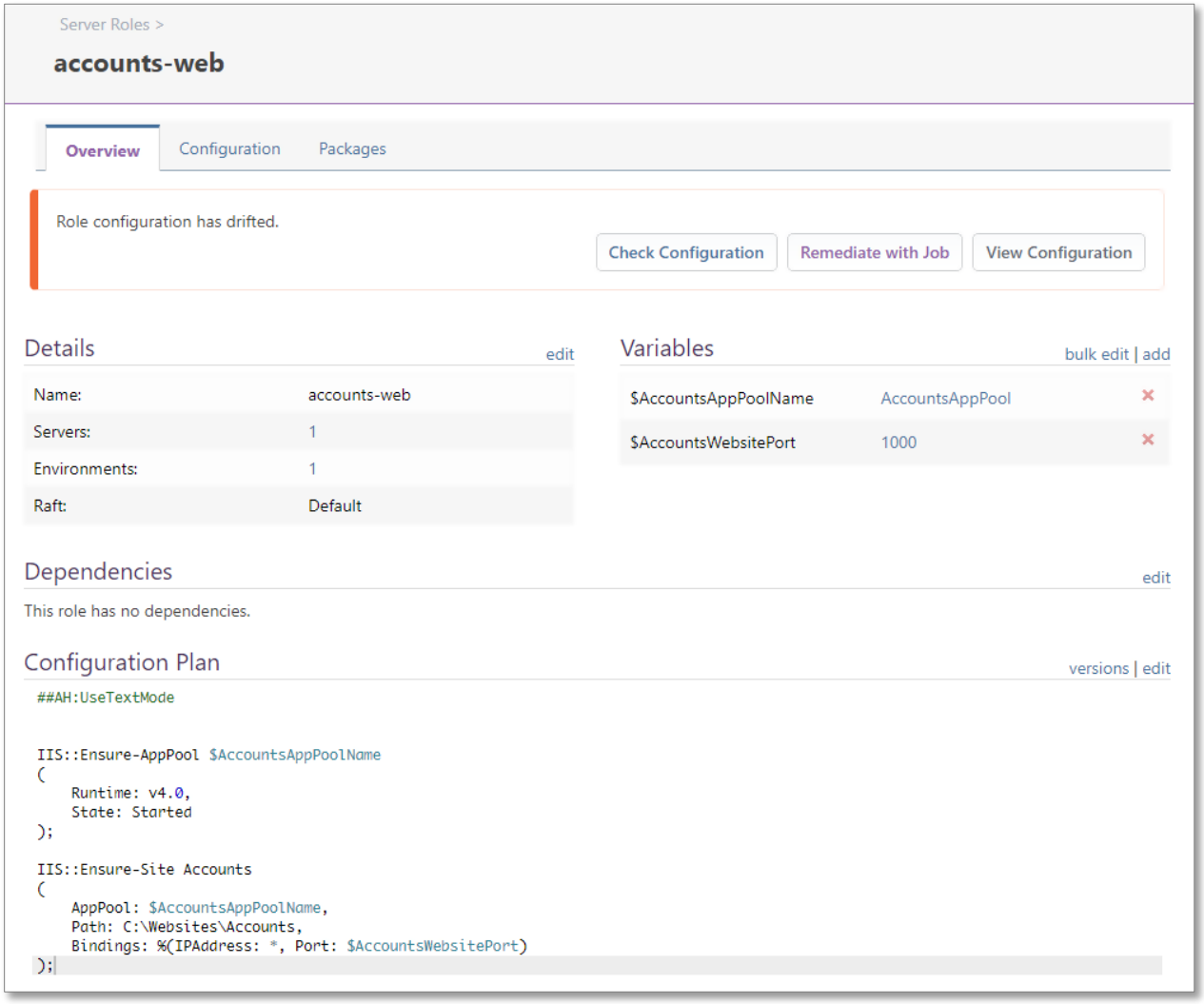
Otter notes that “Configuration has drifted” for this server because the specified application pool or site under the “Configuration Plan” does not exist on that server. This is detailed further under the “Configuration” tab:



By choosing the option to “Remediate with Job,” Otter will create both the application pool and site on the test server. Alternatively, configuring the server to “automatically remediate” will fall more in line with more traditional IaC tools by installing the missing resources when detected.

Changing Configuration without CI/CD

While changing the configuration of a single server is interesting, it is much more interesting to propagate these changes across multiple servers and environments. In Otter, this is achieved with server roles. Instead of declaring the infrastructure directly on the server itself, a new role is created and the infrastructure definitions are assigned:



An interesting note from this role configuration is the usage of variables to define the application pool name and its associated site's port. These configured values act as the default, but any server associated with this role may override that value simply by adding a variable value with the same name. A common example of a port change across environments like this is when SSL is enabled on a testing, staging, or production server, but not a development server.

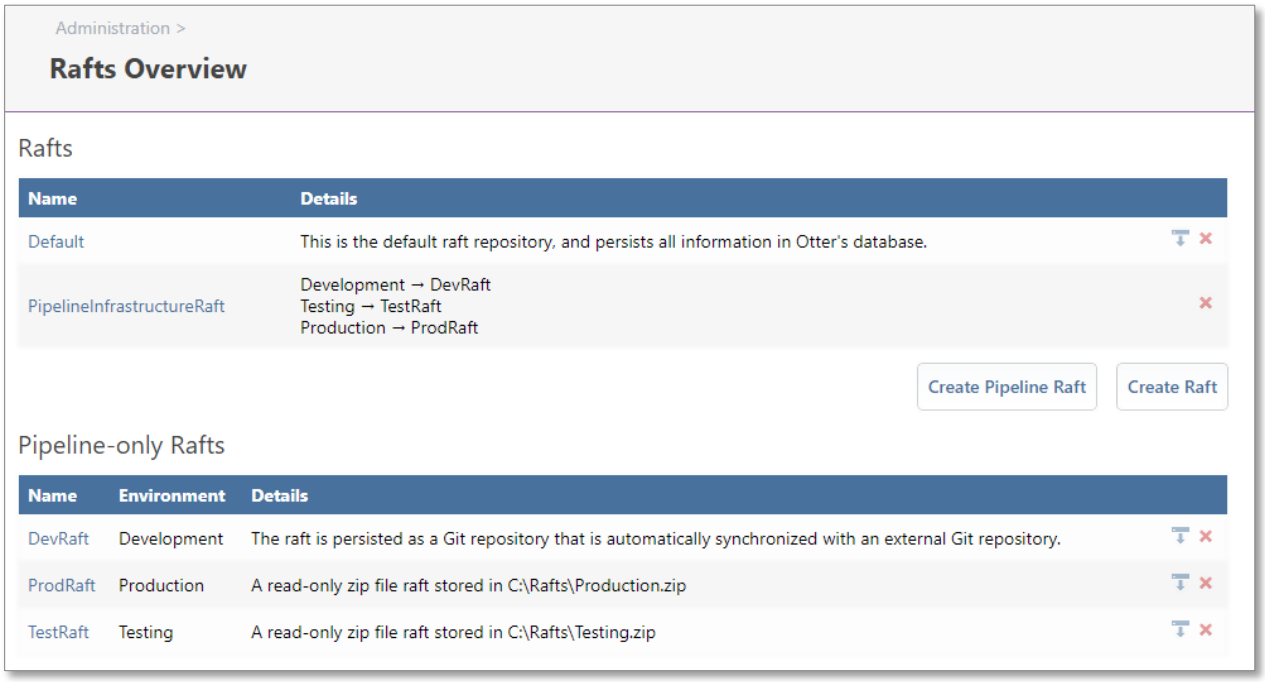
With Otter alone, a pipeline can be mimicked by applying a server role to a server in a later environment upon deployment to that stage of the pipeline. This is effective if the role is defined correctly on the first attempt at a release, but as both software developers and release managers are aware, this sort of perfection is hardly ever a reality.

If, for example, the “pipeline mode” of the application pool were required to change from Classic to Integrated mode, updating the server role’s configuration would update all servers associated with that role regardless of testing in earlier environments. While one small change like this does not typically cause infrastructure problems, several small changes can make it exponentially more difficult to track changes and tests.

CI/CD Set-up: Infrastructure Pipelines with Otter + BuildMaster

By allowing BuildMaster to orchestrate Otter changes within the context of its own pipelines, a complete CI/CD pipeline can be realized without slowing the cycle of development or deployment. At a high-level, the process inherits the benefits of Git rafts for development environments (the equivalent of CI for infrastructure) and artifact creation and promotion for later environments (the equivalent of deployment pipelines). The artifacts themselves are read-only zip file rafts, becoming associated with a release and promoted through pipeline stages as the release cycle dictates while keeping infrastructure in line with application code.

The first step of the setup involves creating a set of “pipeline-only” rafts in Otter and then associating them with a pipeline raft, pointing to a raft depending on environment:



The “DevRaft” in this example is a Git raft and represents the infrastructure of the Development environment. The other two are read-only zip file rafts.

From the BuildMaster standpoint, setup involves configuring an application that captures the infrastructure from Git at a certain point in time into an artifact (a zip file). An example pipeline JSON for this application is:

```
{
  "Name": "Infrastructure-CI-CD",
  "Description": "Pipeline that captures infrastructure from Git, represented by
the DevRaft in Otter",
  "Color": "#9163aa",
  "EnforceStageSequence": true,
  "Stages": [
    {
      "Name": "Build",
      "Targets": [
        {
          "PlanName": "Capture Git Infrastructure",
          "EnvironmentName": "Build",
              "DefaultServerContext": "Specific",
              "ServerNames": ["localhost"]
        }
      ]
    },
    {
      "Name": "Testing",
      "Targets": [
        {
          "PlanName": "Deploy",
          "EnvironmentName": "Testing",
              "DefaultServerContext": "Specific",
              "ServerNames": ["ottersv1"]
        }
      ]
    },
    {
      "Name": "Production",
      "Targets": [
        {
          "PlanName": "Deploy",
          "EnvironmentName": "Production",
              "DefaultServerContext": "Specific",
              "ServerNames": ["ottersv1"]
        }
      ]
    }
  ]
}
```

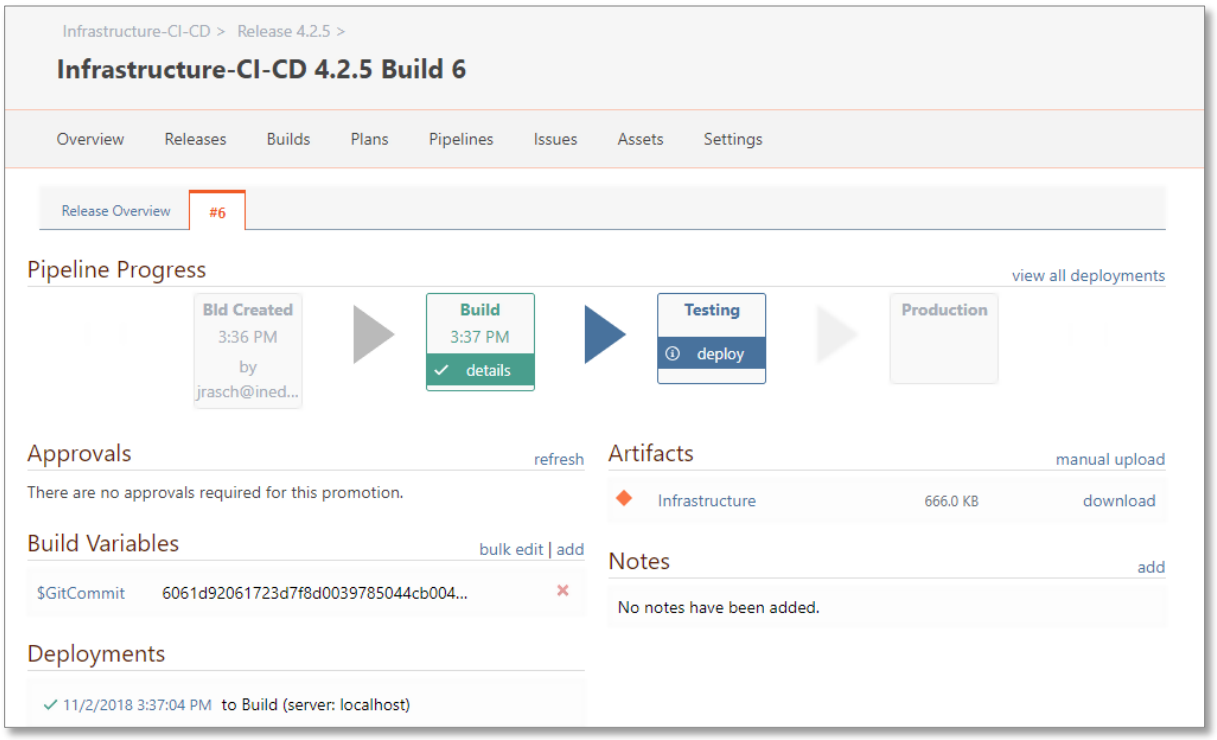

The “Capture Git Infrastructure” plan is defined as:

```
GitHub-GetSource
(
    Credentials: GitHubInedoBuilds,
    Organization: Inedo,
    Repository: OtterRaftTest,
    DiskPath: ~\Infrastructure,
    Branch: infrastructure-ci-cd,
    CommitHash => $commit
);

Set-ReleaseVariable GitCommit
(
    Value: $commit,
    Build: $BuildNumber
);

Create-Artifact Infrastructure
(
    From: ~\Infrastructure,
    Exclude: .git**
);
```

Now, when a build of this application is created in BuildMaster, the infrastructure that was defined in Git becomes an artifact along with the Git commit it was pulled from:



This build can now be deployed through the pipeline using a “Deploy” plan defined as follows:

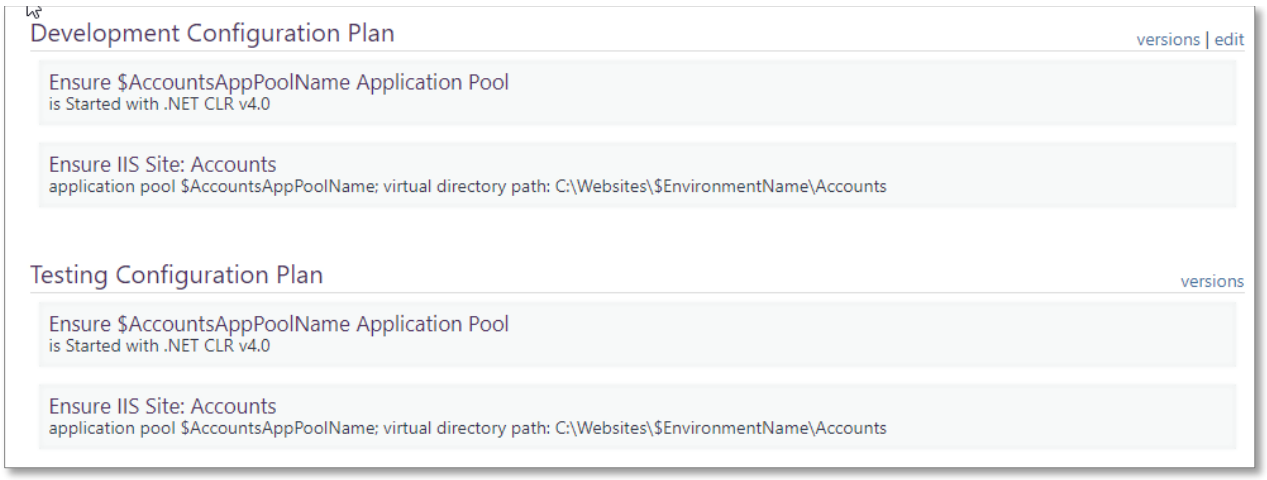
```
Deploy-Artifact Infrastructure
(
    To: C:\Rafts,
    DeployAsZipFile: true
);

Rename-File
(
    From: C:\Rafts\Infrastructure.zip,
    To: C:\Rafts\$EnvironmentName.zip,
    Overwrite: true
);
```

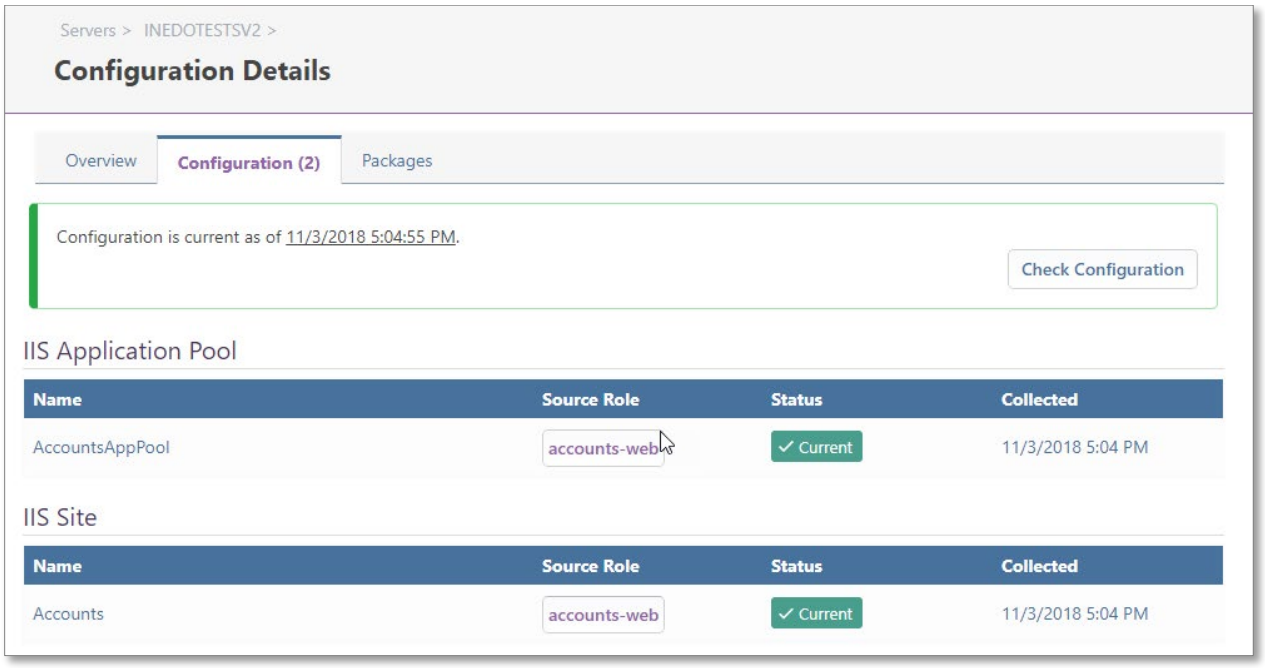
In Otter, we set the “accounts-web” role to use the “PipelineInfrastructureRaft” created earlier, and infrastructure is now ready to be promoted through a pipeline.

Changing Configuration with CI/CD

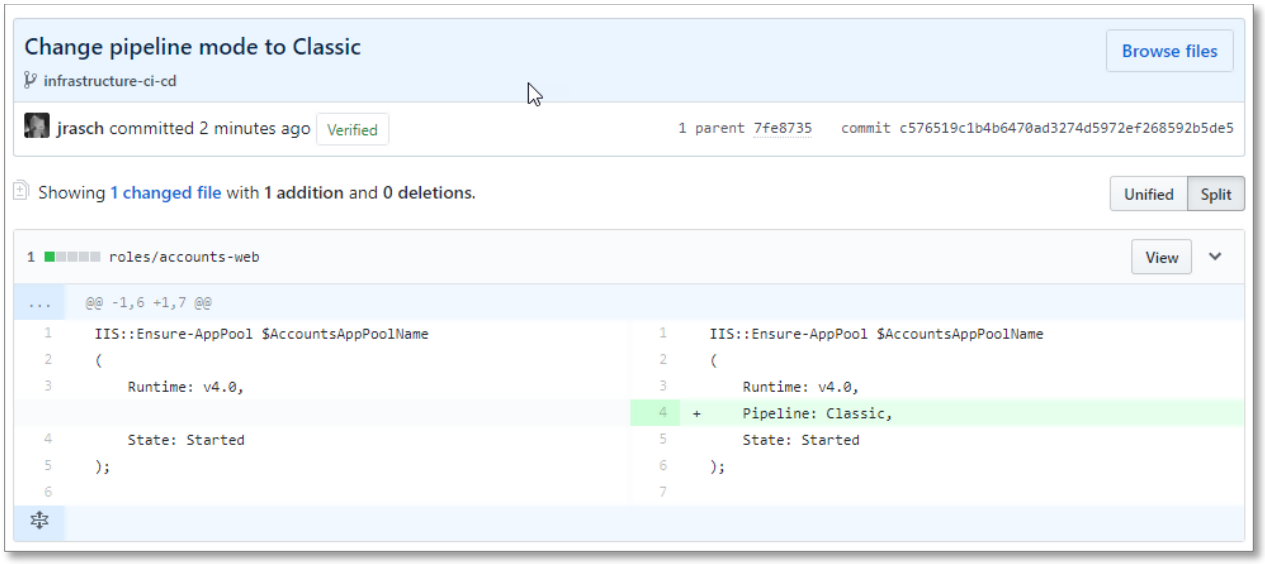
In the following example, we demonstrate the process of applying a configuration change to an application pool and IIS site through a pipeline. First, we will note the role’s specified configuration is the same for the Development (i.e. direct from Git) and Testing (i.e. from the zip file) pipeline rafts:



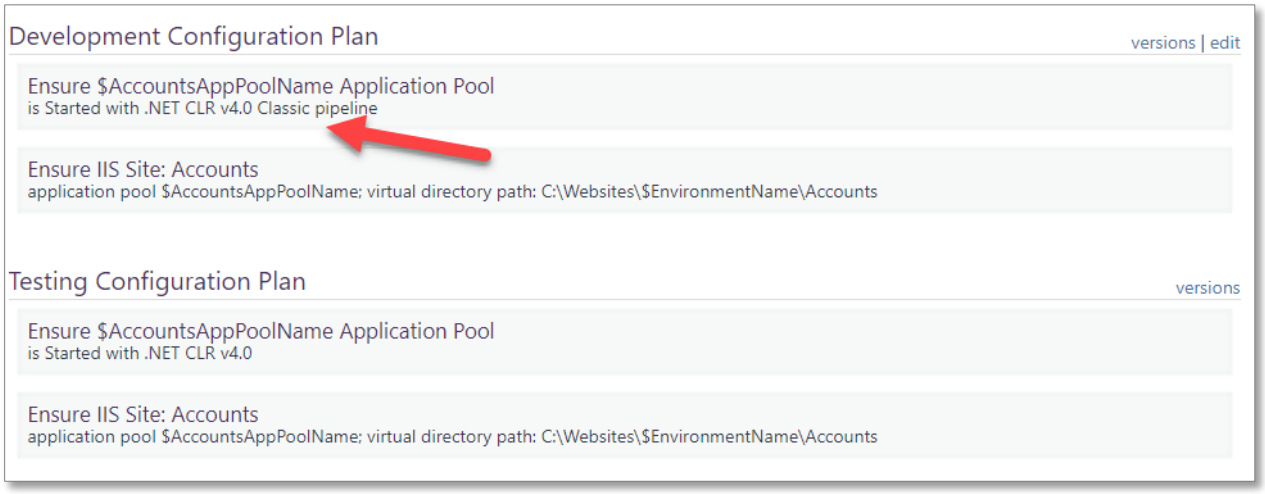
Second, we will note that the test server is up-to-date with the Testing configuration of the role:



Now we will make a configuration change directly to the Development environment, this time by editing configuration within Git instead of Otter:

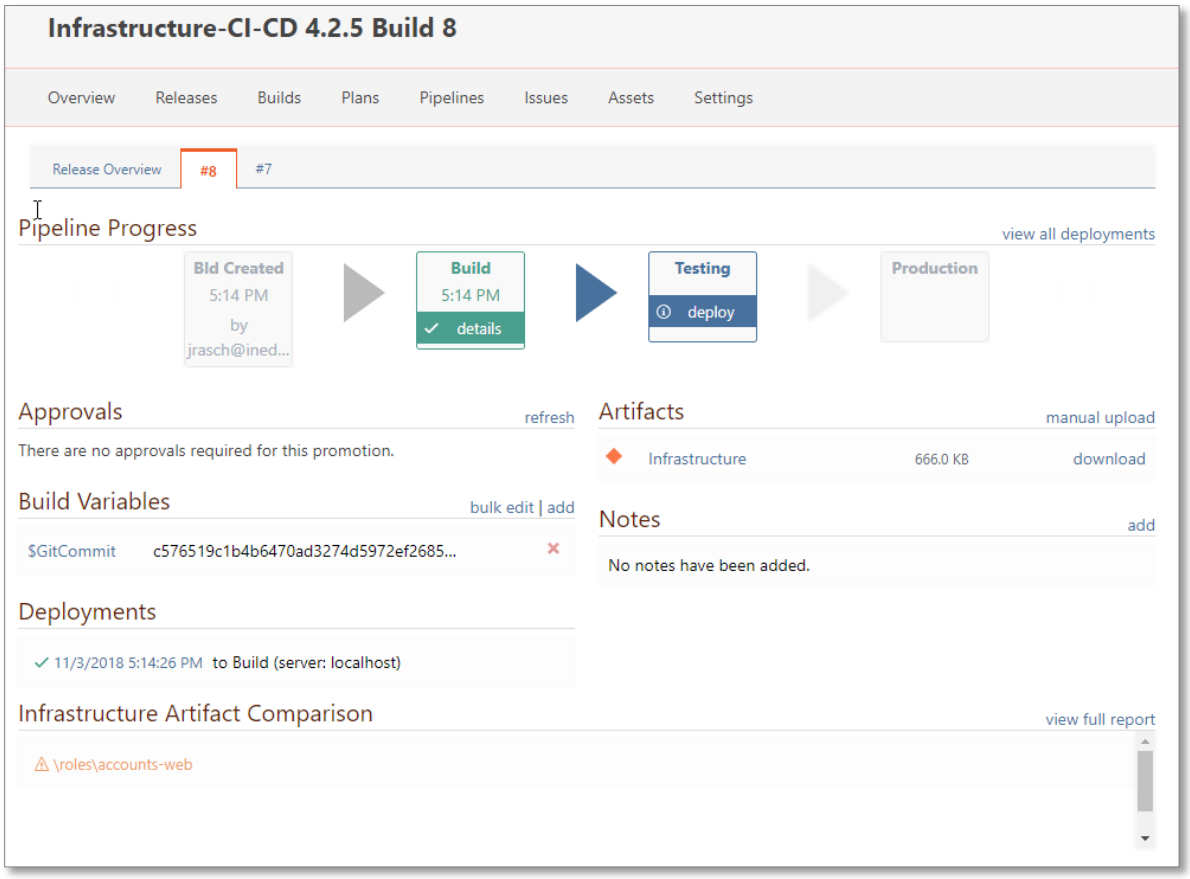


Once committed, this change is immediately reflected in Otter itself on the “accounts-web” role:

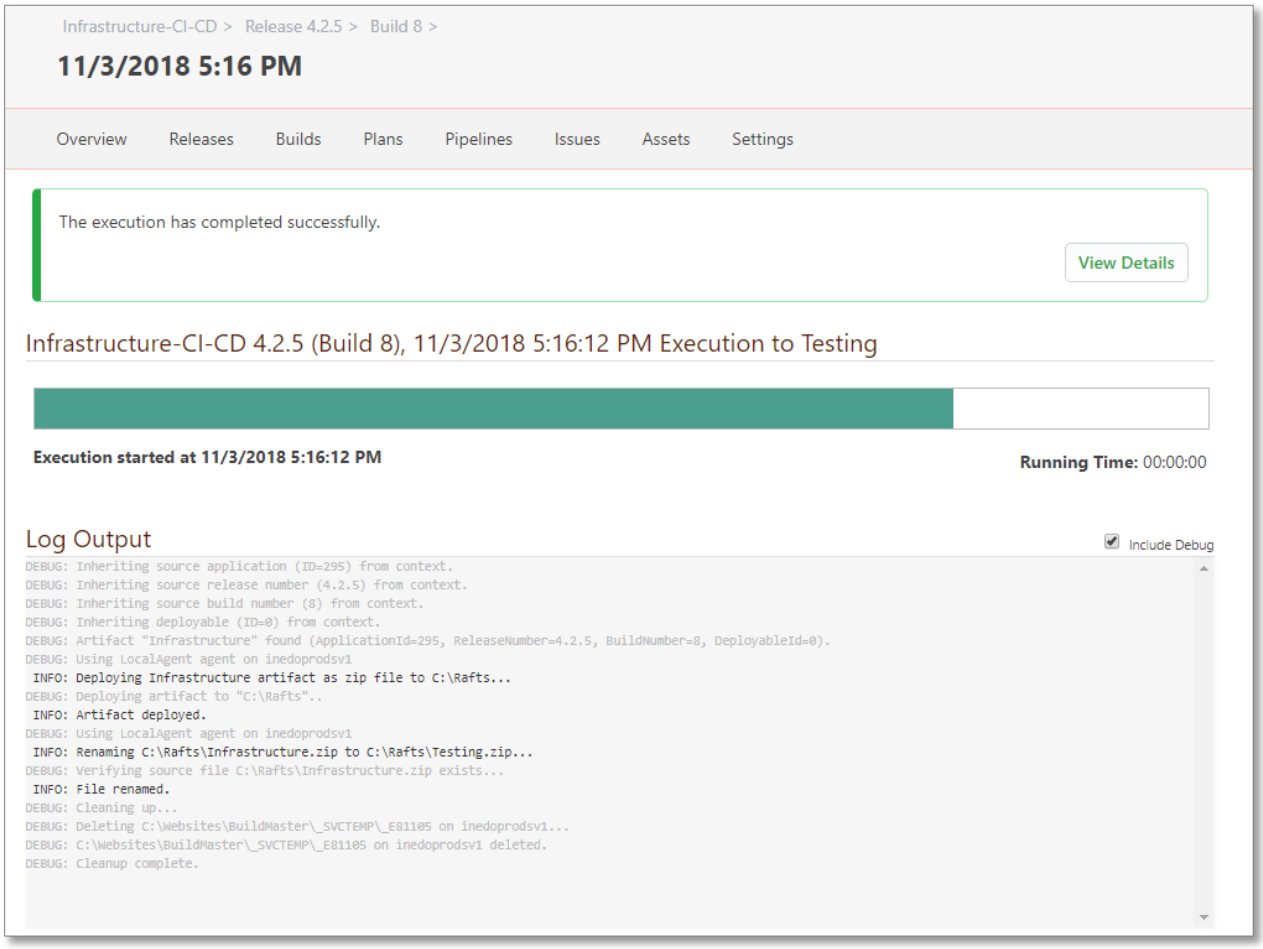


Note that the Testing configuration plan has not changed, and thus, the server associated with the Testing environment still appears as non-drifted.

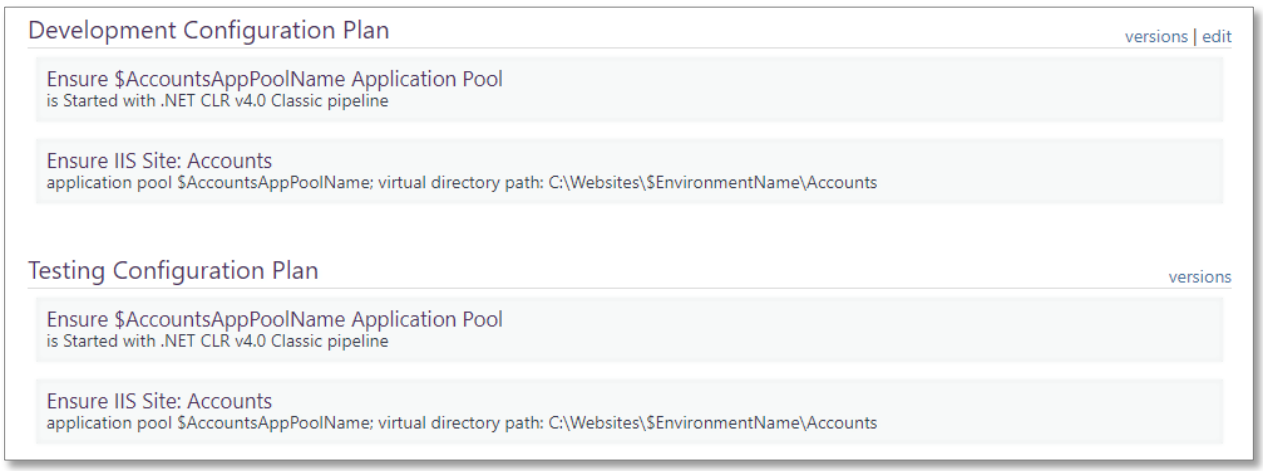
Now we will capture this infrastructure configuration by simply creating a new build in BuildMaster:



And, we will apply it to the testing environment by deploying to the Testing stage:

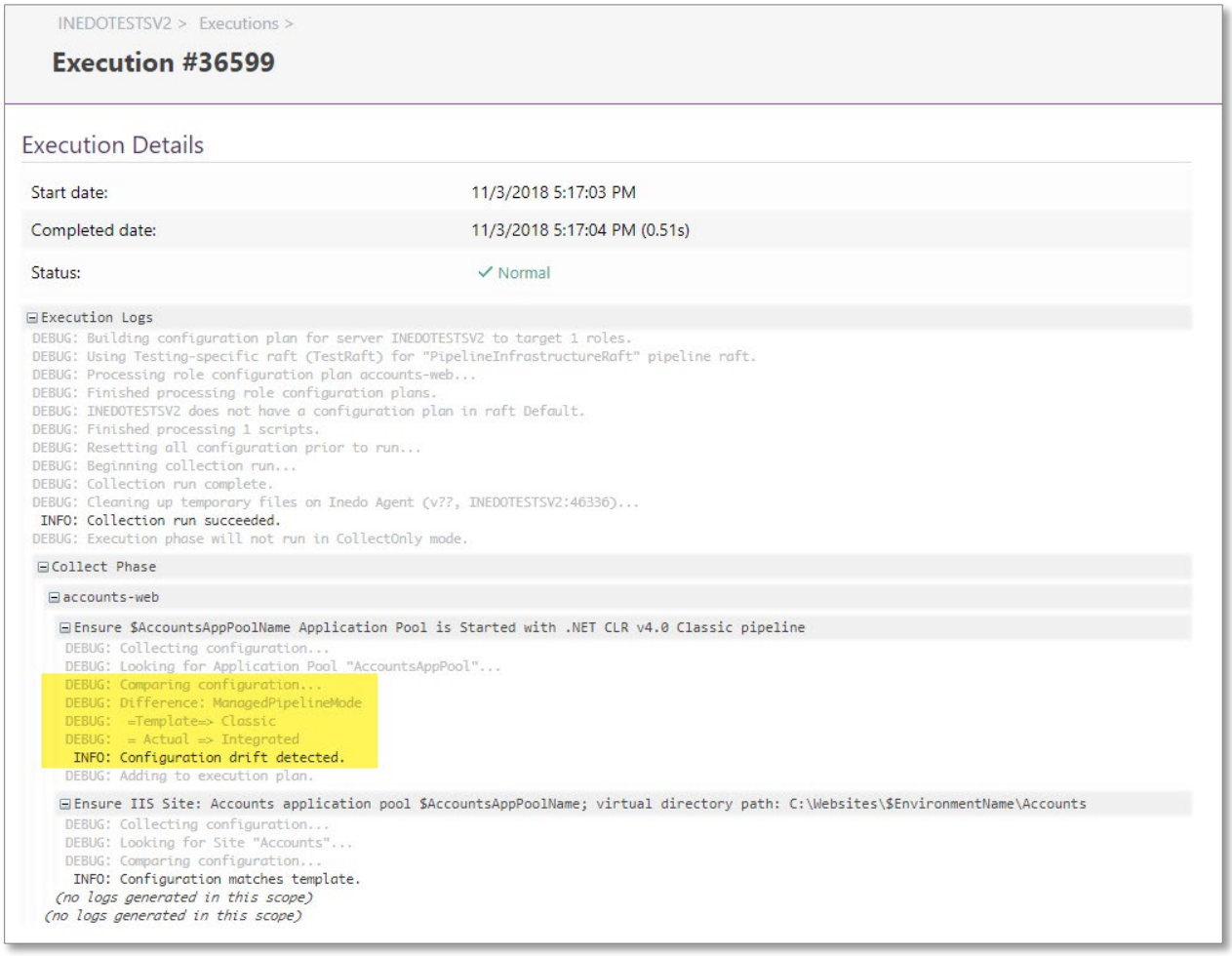


Referring back to Otter, the Testing configuration plan now matches the configuration plan of the Development environment:



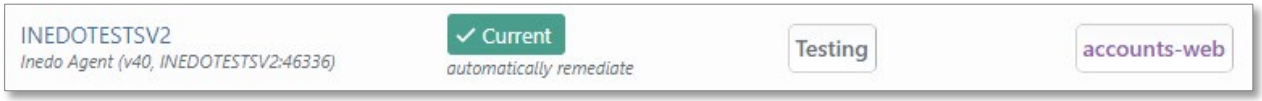
Further, any server associated with this role in the Testing environment is now considered drifted because the application pools as configured on the server are set to Integrated

mode as confirmed by both the server role and server overview pages. The collection log also details the drift:



As mentioned earlier in the setup instructions, we could also change the server's auto-remediation settings from "Report Only:"

Changing them to "Automatically Remediate," Otter remediates the server without any intervention required from the user once the infrastructure artifact is promoted to Testing:

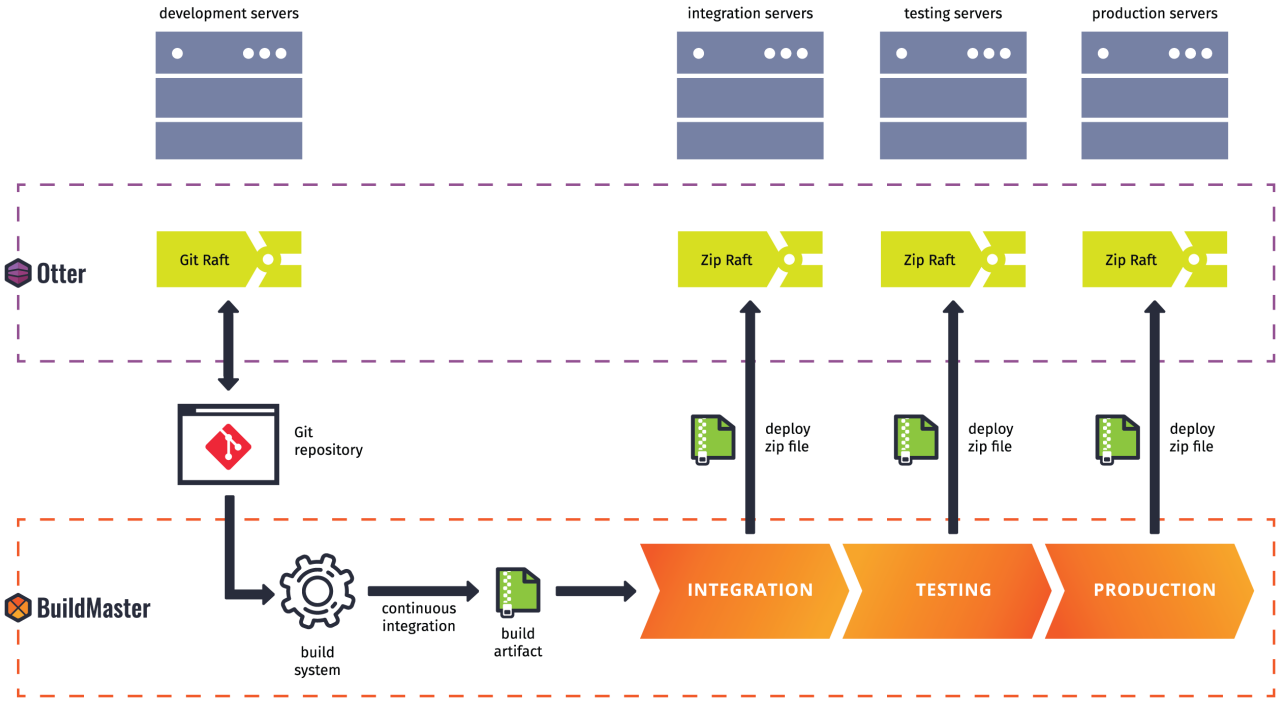


Next Steps

This guide introduced the following topics surrounding CI/CD for infrastructure:

- How and why CI/CD has been so successful for applications changes
- Infrastructure and configuration changes as a new bottleneck
- Challenges with CI/CD for infrastructure and configuration changes
- How to overcome challenges and implement CI/CD for infrastructure
- How to implement CI/CD for Infrastructure with Otter + BuildMaster

CI/CD for infrastructure is an advanced practice. If you're not using a CI/CD tool for application delivery like BuildMaster or a configuration management tool like Otter, you should start implementing and work your way toward continuous improvement.



Remember, you can implement everything using Otter Free and BuildMaster Free editions. Installing takes just a few minutes. Get started today!


inedo