# Configuration Management

**This section is dedicated to the topic of Configuration Management, with an emphasis on related topics, such as automation of CI/CD.**

**A working definition of API Configuration Management is the set of practices for managing the runtime state of API software applications.**

## API Configuration Management

Applications, including those of the API variety, typically consists of one or more databases, web servers, application servers, reverse proxies, load balancers, and other capabilities that work together at runtime to implement a robust working system.

Environment Configuration Management is the process and tools for managing all of the artifact revisions as a set of compatible, functioning components. CM guarantees that Revision α of one component works with Revision α of another component and Revision α of yet another.

### Configuration Management Automation

Prior to the capability of automated configuration management products, the responsibilities of system engineers typically consisted of the creation of items such as scripts, manually maintained files consisting of service and technical endpoints, port numbers, namespaces etc. The tedious nature of this process resulted in the increased complexity of a system as well as the configuration management process itself.

Increased infrastructure configuration management automation evolved because of the needs of processes such as artifact versioning, environment changes and the need for coordination between developers and system engineers.

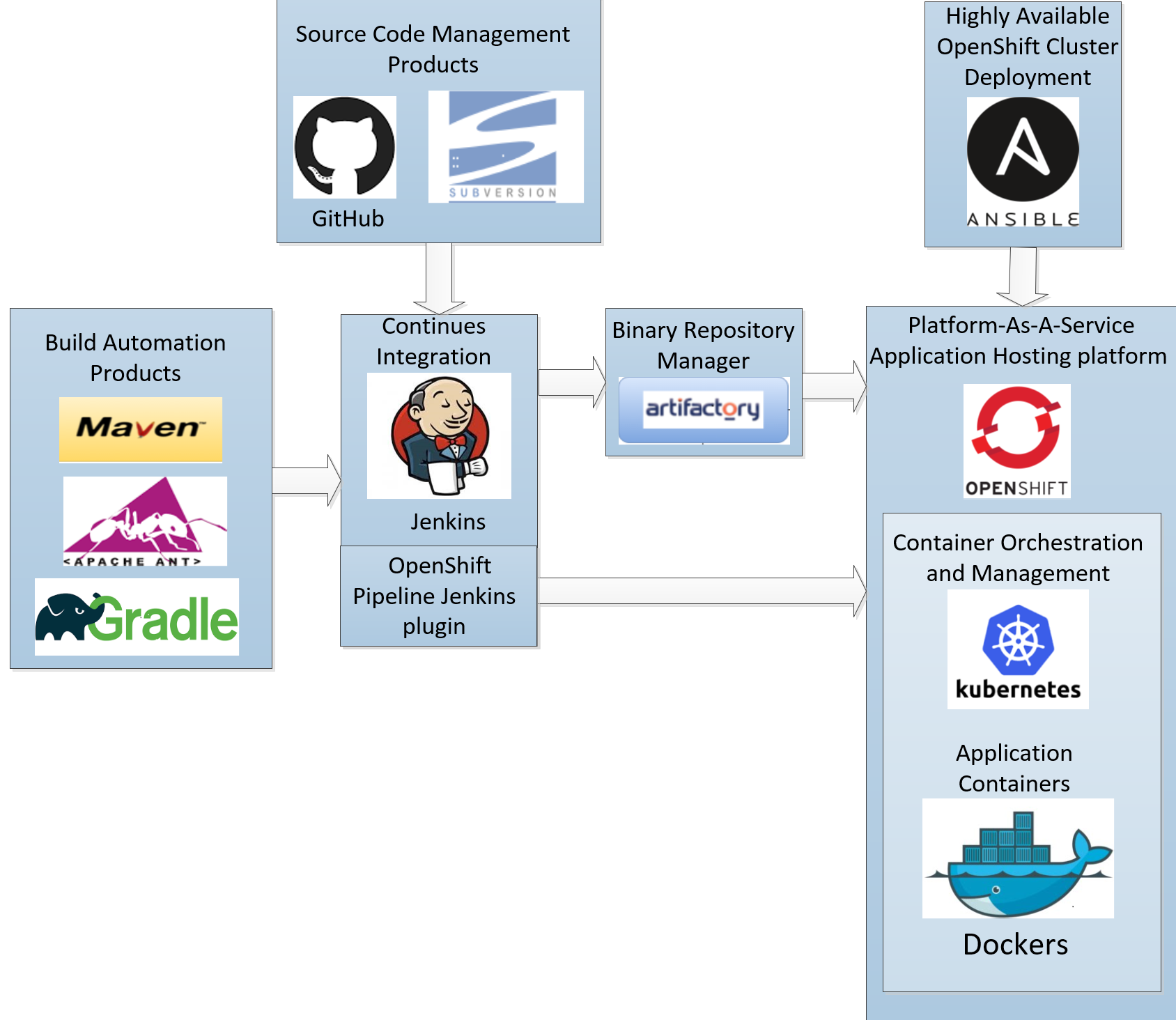
This section describes a conceptual example of an automated configuration management eco-system using a suite of automated products.

The following diagram depicts products that have gained wide acceptance in the configuration management space. There are other potential solutions, but the products shown here are common. Again, API development and configuration management can be implemented using this component example, but a non-API effort can also utilize this model. The diagram depicts the connectivity between components using arrows. Because this is a conceptual diagram the arrows are implemented utilizing plugin components. For example, the integration between GitHub and Jenkins utilizes the Github/Jenkins Plugin. This plugin facilitates the integration of Jenkins with Github projects.

A point should be raised regarding the Configuration Management Automation subject matter covered in this section: The creation of a complex automation eco-system, as detailed in the following diagram, is the culmination of design efforts by many senior technical personnel. Typically, an API developer would not be part of this technical effort, but the intent is to provide the developer with an understandable view of the overall automation eco-system integrations and flows.

(Editor’s Note: Standards review stops here)

Figure 1 Configuration Management Automation Example



Short high-level descriptions (mostly of the marketing variety) of the products depicted in the above diagram follow. Certain component groupings contained multiple product choices. A grouping that have multiple choices will contain a reference to the desirable component.

**Build Automation Products**

* Maven
* Ant
* Gradle

Gradle and Maven are more recent build automation products. ANT is a product that has been in use for several years. Maven has a larger user base as compared to Gradle and is the recommended product. The *ASG\_API\_Playbook\_08.00 Developer Setup Section* has a detailed description of Maven.

{blankline}

**Source Code Management Products**

* Git/GitHub
* Subversion (SVN)

Git is a distributed version control system that has many enhanced features like branching and merging. SVN is a centralized system that has been is use for some time.

Git is not better or worse than SVN, it's just different. If there is a need for Offline Source Control and the willingness to spend some extra time learning it, then Git is a good choice. But if there is a need for a strictly centralized Source Control then the simplicity and excellent Windows based tooling of SVN works well.

Because of the growing acceptance of Git and GitHub, Git is the recommended product.

**Continuous Integration**

* Jenkins

Jenkins is an open Continuous Integration Source Automation platform. Organizations can accelerate the software development flow by automating the steps of the process. Jenkins manages and controls software delivery processes throughout the entire lifecycle, including such tasks as build, document, test, package, stage, deployment, and static code analysis.

* OpenShift Pipeline Jenkins Plugin

OpenShift provides a Jenkins Container as the CI/CD tool to run on an OpenShift cluster. This Jenkins Container is used to set up a configuration pipeline to deploy applications to an OpenShift cluster. This is an integrated solution that is different from running Jenkins outside OpenShift that deploys applications to an OpenShift cluster.

{blankline}

**Binary Repository Manager**

* Artifactory

Artifactory (short for Artifact Repository) is a product that serves as a binary repository manager. The binary repository is a natural extension to a source code repository. It will store the outcome of a build process sometime known as artifacts. For example, artifacts in the Java world are jar, ear, war etc.

{blankline}

**Highly Available OpenShift Cluster Deployment**

* Ansible

Ansible is product that automates software provisioning, configuration management, and application deployments. It automates cloud provisioning, configuration management, application deployment, intra-service orchestration plus other IT needs.

{blankline}

**Platform-As-A-Service Application Hosting Platform**

* OpenShift

OpenShift Container Platform offers a full enterprise enterprises level control over Kubernetes environments, whether they’re on-premise or in the public cloud, this provides the capability to build and run applications anywhere.

{blankline}

**Container Orchestration and Management**

* Kubernetes

Kubernetes is a product for automating deployment, scaling, and management of containerized applications. It groups containers that make up an application into logical units for easy management and discovery.

{blankline}

**Application Containers**

* Dockers

Docker is producer that performs operating-system-level virtualization, also known as "containerization". It helps create and deploy software within containers. It’s a collection of tools that help you “Build, Ship, and Run any App, Anywhere”

## API Versioning

A viable versioning strategy for software artifacts is a crucial consideration within configuration management. This section will detail versioning, when deploying APIs, using the RESTful architecture and its implementation from a HTTP Request and Response perspective.

### When to Version

APIs only need to be up-versioned when a breaking change is made. Breaking changes include:

* A change in the format of the response data for one or more calls
* A change in the response type (i.e. changing an integer to a float)
* Removing any part of the API.

Breaking changes should always result in a change to the major version number for an API or content response type.

Non-breaking changes, such as adding new endpoints or new response parameters, do not require a change to the major version number. However, it can be helpful to track the minor versions of APIs when changes are made to support customers who may be receiving cached versions of data or may be experiencing other API issues.

### Versioning Guidelines

There is substantial discussion documentation on the web regarding RESTful versioning. This document will detail **two** options that for the most part have been overwhelmingly referenced in these discussions:

* URI Versioning – version the URI space using version indicators
* Media Type Versioning – version the Representation of the Resource

Pros and cons of each option will be noted.

### URI Versioning

The first recommendation addresses how to handle versioning if the development community views the URL as the most important facet which make the URL the center of the versioning efforts. Developers who are familiar with versioning with WS-\* Web services tend to be more comfortable with changing the URL when versions change.

Like most other problems with REST, a common flawed approach to API versioning stems from misunderstanding of the URI. This popular way simply adds a version number as a segment in the resource identifiers

Adding, removing, reordering, and renaming fields constitutes a breaking change for a client which philosophically is a departure from REST. Creating a new version in this scenario, changes the URL structure. The resource object version for 1.0 and 2.0 looks like this:

* http://www. mycompany.com/myapp /resource1.0
* http://www. mycompany.com/myapp /resource2.0

In the above examples it is most likely the behavior of the resource that changes and not the actual resource. In other words, it is the contract that changed. The two identifiers are used to access the same resource. This is the most common argument why this approach is somewhat problematic but it is simple and will never break any cache constraints.

By convention, the default value for format is json. Examples:

* Request the resource as XML: http://www. mycompany.com/app/1.0/resource ?format=xml
* Request the resource as JSON: http://www. mycompany.com/app/1.0/resource ?format=json

Some developers tend to prefer versioning by URL which allows them to determine which version of the service is in use at a glance. From a purest standpoint the URI in REST should represent the resource structure only. It could be argued that a version is not a resource but an attribute of the resource.

### Media Type Versioning

The REST API would make use of custom vendor Multipurpose Internet Mail Extensions or MIME media types instead of generic media types such as application/json. It is these media types that are going to be versioned instead of the URIs. The media type versioning philosophy makes use of the MIME sub type of vnd which is vendor-specific. The Internet Assigned Numbers Authority (IANA) is the official authority for the standardization and publication of these vendor classifications.

If the API is to be used internally only, the vnd MINE type should not be an issue. If an external presence is desired, thought should be given with registering it with the IANA This is a standard way for sharing out-of-band information with potential clients.

#### HTTP Request Accept Examples:

* **Accept**: application/vnd. mycompany.myapp.resource-v1+json

or using a v=1 qualifier:

* **Accept**: application/vnd. mycompany.myapp.resource+json;v=1

#### HTTP Response Content-Type Examples:

* **Content-Type**: application/vnd. mycompany.v1+json

What is important to understand here is that the client makes no assumptions about the structure of the response beyond what is defined in the media type. Therefore, generic media types are not ideal as they do not provide enough semantic information and force the client to use additional hints to process the actual representation of the resource.

The Accept header in the request tells the server the content types that the client is expecting in the response body.

Content negotiation preserves clean set of URLs but there is still the task of having to deal with the complexity of serving different versions of content somewhere. This burden tends to be moved up the stack to the API controller infrastructure which become responsible for figuring out which version of a resource to send. The result tends to be a more complex API as clients must know which headers to specify before requesting a resource.

### URI Versioning and Media Type Versioning Evaluation

If the development community is more RESTful architecturally minded and aware, REST API versioning should be accommodated by using version data representations in the MIME types in the HTTP Application Accepts. If development community views the URL as the most important facet then changing the URL should be used when versions change.

Both mechanisms are valid. You need to know your consumer to know which path to follow. In general, working in an enterprises minded environment tends to point developers towards REST versioning using the MIME philosophy.

## GitHub Integration

Git is a distributed software configuration management (SCM) version control system that as the name implies facilitates the versioning of software artifacts. GitHub currently hosts the source code for numerous open source projects.

An in-depth discussion on the pros and cons of Git compared to centralized source code control systems will not be a part of this discussion as Git and GitHub is the VA’s recommended SCM alternative.

Because of the recommendation of GitHub as the VA alternative, a discussion of the GIT Branching and Merging capabilities will be discussed. Branching and Merging is an integral component of API software configuration management.

### The GIT Command Line

There are a lot of different ways to use Git. There are the original command-line tools, and there are many graphical user interfaces of varying capabilities. The command line is the only place where all Git commands can be executed. Most of the GUIs (e.g. Bit Bucket) implement only a subset of Git functionality for simplicity and the choice of a graphical client is a matter of personal taste. For purposes of clarity only GIT command lines features will be discussed in this section of the document.

### GitHub Branching and Merging

It can be argued that Git changed the way developers think of merging and branching. From the classic CVS/Subversion world where merging/branching has always been considered a bit daunting and something you only do when it was necessary.

With Git, merging and branching actions are simple, and they are considered one of the core parts of a daily development workflow, for example, in the world of CVS/Subversion, branching and merging is treated as an advanced concept, while in Git they are considered basic concepts.

This section of the document will cover GitHub Branching and Merging commands at a somewhat low level using Git command lines as part of the discussion. A detailed philosophical discussion of Branching and Merging is better left to the configuration management strategies of individual development teams.

### GitHub Branching

Software development generally occurs using multiple branches in parallel. For example, a software artifact may be under development/modification by multiple developers on different branches.

Typically, there are multiple types of branches utilizing various naming conventions:

* **Master Branch -**  Publicly released production-ready code.
* **Release Branch -** Branches that are being prepared for deployment. but still internal to the deployment effort.
* **Project Branches** - Branches in development for pre-release projects, such as partner deployments. Branched that are at a lower level then a Release Branch artifact.
* **Development Branch**: Branches that contain in-progress artifacts that are being worked by development staff.

These branches are arranged in a liner sequence where the Development Branch migrates to the Project Branch and so on upwards. This linear sequence simplifies the *merge/release strategy* (i.e. Develop to Project to Release to Master)

The *merge/release strategy* alluded to above means that when code is ready to be moved up the branch food chain a "forwards merge" along the direction of this chain is performed. Conversely a "backwards merges" are performed on a regular basis to bring changes (e.g. hotfixes) from the higher level branched back down to the lower level branches. Backwards merges can happen any time, but should be regularly scheduled.

Choosing where to branch off from and target a pull request depends on an assessment of three factors: how urgent the fix/feature is, how risky (in terms of introducing other bugs) the required changes are, and how long it will take to develop. A good heuristic is to see how urgent the code that is being committed versus how risky or potentially breaking it is:

#### GitHub Branching and Related Command Lines

Simply stated a branch is required when a feature hosted in the high-level branch must be modified and/or added. A branch which will host the development of the new feature, must be created from an upper level branch, which is typically the Master. This section will focus solely on the branch creation process only. Subsequent process flow commands such as commits and check-ins are not part of this discussion. The following are typical steps used in creating and removing a branch:

1. Creating a new branch:

Execute the Git ***branch*** <new branch name**>** followed by Git ***checkout*** <new branch name***>.*** These two commands can be combined into one by using the checkout command with the -b switch - git ***checkout -b*** <new branch name**>.**

1. Removing a branch

Branches are typical removed after their contents have been merged to a higher-level branch and is good practice. Execute the Git branch command with the -d switch - Git ***branch -d*** <branch name**>.**

### GitHub Merging

GIT merging facilitates consolidates the migration of a forked branch or branches back to a common branch. The Git ***merge*** command accomplishes this task.

The current branch will be updated to reflect the merge, but the target branch will be completely unaffected. GIT ***merge*** is often used in conjunction with Git ***checkout*** for selecting the current branch and Git ***branch -d*** for deleting the merged branch target branch

Typically, Git ***merge*** is used to combine two branches. The following examples will focus on this branch merging pattern. Git ***merge*** takes two commit pointers, usually the branch tips which are the last commit or most recent commit on a branch, and will find a common base commit between them. Once Git finds a common base commit it will create a new "merge commit" that combines the changes of each queued merge commit sequence

Here is the scenario: a branch containing a new feature that is based off the master branch. This feature branch needs to be merged back into the Master Branch. Invoking this command will merge the specified branch feature into the current branch, which in this case is Master. Git will determine the merge algorithm automatically.

#### GitHub Merging and Related Command Lines

Before performing a merge, there are several preparation steps needed to ensure the merge is performed correctly.

1. Confirm the receiving branch of the merge:

Execute Git ***status*** to ensure that HEAD (think of the HEAD as the "current branch") is pointing to the correct merge-receiving branch. If needed, execute Git ***checkout*** <receiving branch> to switch to the receiving branch. In this case execute git ***checkout*** ***master***

1. Fetch latest remote commits:

Insure that the receiving branch and the merging branch are up-to-date with the latest remote changes. Pull the latest remote commits by executing Git ***fetch***. Execute Git ***pull*** once the fetch is completed to ensure the master branch has the latest updates

1. Perform the merge:

Once steps 1 and 2 have been taken place, a merge can be initiated by executing Git ***merge*** <branch name> where <branch name> is the name of the branch that will be merged into the receiving branch which in this scenario is Master.

#### Resolving GitHub Merging Conflicts

If the two branches both changed the same part of the same file, Git won't be able to determine which version to use. When this a situation occurs, it stops right before the merge commit so the conflict can be resolved manually. When a merge conflict is encountered, running the Git ***status*** command shows which files need to be resolved.