Delivery Automation Framework - Overview

> This is an overview of some technology, concepts and principals that can go into building a scalable enterprise scoped delivery automation framework. In the context of this document, the term delivery refers to the provisioning and deployment of the business software products, supported 3rd party technologies and various cloud objects such as machine instances, services, and other infrastructure components as well as ongoing operationalization of each.

Basic building blocks

The following section highlights some basic building blocks needed to design a scalable system that should support multiple aspects of the business.

Source control (config) management system

Integrating your configuration into a source control system is essential. All configuration should be managed in a true source management system such as github. The configuration should be realized programmatically by the automation tooling. No manual moving around of configuration should be necessary.

Following this principle, the configuration becomes code and is handled just like any other software project with respect to test and release process.

Source control (product) management system

Implementing a good product delivery system is highly dependent on good source control management practices. All product specific and configuration code should be separate but managed software repositories.

Engineering best practices should be utilized at all phases of the SDLC. From branching strategy and versioning to quality, deployment and release strategy. Without these and other basic engineering principles and best practices in place, it is difficult to build a scalable and sustainable product delivery system.

Build system and pipelines

Continuous delivery is goal of most software organizations in today's would. Especially when it comes to hosting Saas or other software in the cloud or cloud native environments. "CD" cannot be acheived without strict control and quality gates in place at various phases in the SDLC.

Environment (config) promotion strategy and artifact promotion are keep concepts that need to be determined before attempting to go full CD.

Pipelines are the best approach to getting a continuous deliverly SDLC off the ground. Pipelines are the main conduit, or orchestration of various phases of delivering a product without human intervention.

A typical pipeline would be similar to this:

code checkin - run build (unit test/compile) - publish artifact/assets - update config programmatically (the build deploy handoff) - invoke a deployment - run post deploy end to end testing - reporting/coverage etc.. - update config/enable application

Within this sample pipeline many many things happen and all need to be 100% automation. If there is any human intervention, it is NOT a pipeline.

The pipeline is an orchestrated sequence of events with logic and deterministic behavior happening at various phases. This deterministic logic is the codified buisiness and software use cases for delivering a product. Improvement to the pipeline is an ongoing thing and never ends.

Pipelines have multiple injection and exit points to allow flexibility in exercising various phases without running the end to end pipeline each time. For example in a sandbox/dev environment you might not run post deploy End to end tests as those systems may not be available in the sandbox/dev environment. In this case the pipeline needs an alternate injection or exit point.

Orchestration within the CD pipeline in most cases will have a phase that starts an entirely different orchestration. The "deploy" phase for example would consist of a series of events that will have thier own orchestration requirements. For example, provisioning infrastructure as a preamble to deploying code. This is an extreme high level example. A deployment orchestration will have many layers and phases within itself.

More about orchestration is discussion below.

Quality Engineering system

QE becomes a very important basic building block in developing an end to end automation framework. Quality gates throughout all environments before moving to a different environment is very important.

Since automation by its definition does not involve humans, a good quality engineering architecture must be implemented. Within this space, you should have code coverage and analysis, unit test at the component level, syntax checking at a minimum for configurations, end to end testing of all systems that will supports whatever is being delivered, and full integration testing.

There are many frameworks and automation within the Quality engineering segment of SDLC alone.

With a good handle on this, you may as well not attempt to build a product delivery framework.

Even if the automation is being implemented to simply stand up integration systems that exclusivley use 3rd party software, you will need to have very tight integration testing for all you use cases.

Artifact and asset management system

Implementing a true artitact management is another core building block in any delivery framework.

Automation needs to be aware and confident that is can find the assets is needs when it needs them. Using basic file systems, shares and simple file storage will not suffice. Artifacts/assets need to be protected and retreived programmatically with API's and other methods. Imutable artifacts cannot disappear. Storing these in a true artifact repository helps gaurentee this.

Automation such as deployment systems would use api's to retreive assets after being told what to get.

Deployment system

Deployment automation is an extension (phase) if the overall pipeline, if pipelines are in use. The deployment itself is an orchestrated sequence of action and events. Deployment specifications should never be part of the product build. All deployment functionality should be external to any distribuable binaries, packages or deployable components.

Software components such as shared libraries are typically NOT deployable packages. They are usually consumed by another piece of code as a "build time" or "runtime" (aka deploy time) dependency.

These different methods of getting the product/application/service to the final hosted environment are something that should be well throught out as part of the software architecture and workflow use case modeling.

Identify whether something can only be distributed on the back of something else, anytime on its own, or other use cases before designing the deployment strategy.

Basic concepts

One system serves all

Designing and implementing an Enterprise wide system that is scalable is a big challange with a bigger payoff. An automation delivery framework should be flexible to meet the needs of all the various groups and departments that will be supported with such system.

Different enviromments, different product groups, different technology needs, different release cycles and other variable do not require different delivery systems. A delivery system may include, and most likely will include a variety of technologies to suit the needs to these variables and deltas. But the common framework should be scalable enough to extend to the needs of new technologies and new initiatives through the enterprise.

Designing and implementing a Framework to be used a the conduit for provisioning and delivery of product/systems of this scope is not trivial and does require lots of requirements gathering, planning, etc...

However, the learning curve for consumers of the system is far less and not so isolated with respect to tribal knowledge. If an organization has each team run with thier own deliver system, only folks on the team will know the mechanics and thus support becomes tribal knowledge.

Onboarding new engineers becomes a big challange when you have different systems that ultimatley do the same thing, but not in a consistent way.

Naming standards

Name standardization of everything from object tags, artifact names, branch names, source repository names, configuration keys and everything in between is one of the MOST IMPORTANT aspects of building a highly scalable and reusable automation framework.

This should be one of the first things to get sorted out and signoff on from stake holders before starting into the development phase of the automation framework.

Following good naming standards and publishing those standards to the organization is ***crucial.*** Think namespaces. In code (or even DNS) you have two things that are called X. How do you structure the data around X for its consuption. You use namespaces my.X, your.X. Now we have two different implementations of X.

Names should be simple and recognizable, and should tell the viewer what something is, or what prupose it serves etc... Applications called "UI", or 'REST-API" are good examples of very bad names.

In configuration, one method of creating scalable data is to break things down in lower level objects. For example:

System (Top tier representation of a collection of systems/applications etc...)

Group (Mid tier respesentation if a collection of components/applications/services within a particular Suite)

Product (Low tier representation if an application, shared component/lib, service etc...)

Using this method you can have two separate products called APP1 being developed by two separate groups or divisions within one organization. This happens very frequently actually. Teams are not always in full sync with each other at the level of naming and such team specific decisions. So incorporating this into you naming conventions will help avoid any confusion and/or conflict at any time in the future.

Group1.App1

Group2.App1

More discussion about this and how profiles/types fit into this.

Consistent Deployment/Delivery across all environments

This is one of the greatest benefits of an enterprise wide Automation Framework. Delivering infrastructure and product in different environments using the same system avoid a lot of churn and resources and provides a better guarantee that you will not have unforeseen issues when with environment promotion/release of new products and processes.

To acheive 100% consistency requires quite of a bit of use case research and highly parameterized automatioon and configuration. For example in a production scope environment, you may have single purpose servers hosting applications, but in you developement environment, you are stacking applications on multi-homed machine to conserve resources and to speed up lower level component testing. The deltas in thi case should all be parameters to automation or configuration (templating etc...). But the workflow, conduit and underlying tooling should be the same.

Using feature flags/toggles is one method of altering the delivery using the same system.

In addition to the obvious benefit of having consistency in delivery across multiple environments, the onboarding and learning curve is also greatly reduced. Contributors and users of the system only have to learn once and will be able to perform regardless of the environment.

Continuous Integration

CI is a core principle in any good software engineering shop. It is only mentioned here as it is indirectly a required component in any good Automation Framework.

Without code coverage and good test driven development, you will never have the confidence needed to get to CD. To get to CD you have to pass through many gates (environment promotion etc...). You will never have consistency and be able to safely perform repeatable continuous delivering of infrastructure and/or products without knowing that what you are delivering is solid and stable at the component level. Chasing red herrings and hours of resources being needed to find root cause can be greatly reduced if a good CI system with constraints and gates is employed.

Continuous Delivery

CD is one the goals that many Saas shops strive for. Acheiving CD is not trivial. There are many "gates" and valves that need to be in place. Additionally code/asset promotion between environments becomes a major function of a CD system, so must be well thought out.

CD systems are often implemented in or as part of a pipeline. So the same tooling and infrastructure is required such as build system, source management system, deployment system, TDD, testing frameworks, artifact management.

Use case modeling

The first big phase of any automation framework should be to gather requirements and more specifically document all the use cases for anything that will be automated. Designing the configuration structure for 3rd party and in house products should repect the basic principle, create one for many.

If you have a configuration for a productX (a cluster or single entity) and that product is something that may be implemented/deployed by multiple teams/groups within the enterprise, the base configuration should be created as a common base config.

Typically common configs are shared configs, fully rerusable.

Configuration is code

A basic strategy is to treat you configuration as you would a software project. A dedicated software repository, branching and release strategy should all be considered when designing an enterprise scalable system.

Use pull requests, code reviews and have deployment automation be wired to your configuration repository. This is a requirement for Continuous Delivery.

Advanced concepts

Reusable configuration

refer to 'Use case modeling'.

The source control repository that is the single source for the automaiton configuration has to be well thought out and be able to extended by the unknown. Creating re-usable configuration is important for consistency through different environments as well as different implementations of the same product/app/service.

Externalizing implementation specific configuration as well as environment specific configuration is how to achieve 100% reusable configuration. The "state" or "type" etc... is the actual application/product/service or multiple of that you are applying to provisioned instances. Implementation specifics are things like how many nodes, what size nodes, dependencies and dependents.

Only the core organization wide settings should be part of the common configuration. I.E. stuff every implementation gets.

Build in stackable layers or blocks.

Common/shared, then implementation specific, then environment specific.

Extracting environment specific deltas

Separation of environment specific deltas is something that needs to be implemented. This also means that all "common/shared" configuration must be parameterized.

Rendering is a key concept and function of configuration management tools.

Example:

In the Saltstack world, you have two configuration "trees" to work with. The "State" tree and the "Pillar Tree". Typically each of these is mapped to it's own file system or source repository (preferred).

The salt state tree holds all your role/server/type/product/app configuration. Configuration here is the meat of the machine config and should be generic in that the state can be applied to different environments and/or implementations. I.E. if the business decides to support RabbitMq. There should be a single reusable rabbitMQ state that can be consumed by different teams etc... Hence the state must be generic enought to handle that. This is good state design. Always built re-usable states to avoid duplication and conflicts.

For example, if you want to define a web server running product X in your salt state tree, the webserverX.sls salt state content would contain all the actions needed to make a new vm/instance "become" a webserverX node. This could be steps like: install base security config, install dependency package, install webserverX. In this case those three actions together form the 'State" webserverX. Now if team A needs to deploy webserverX in dev and stage environments, as well as Team B needing to deploy webserverX in thier own dev environments, all use the same Salt state, webserverX.sls.

This allows you to manage the state repository just like any other software project. As an example, I have used the branching model where I had a "release" and "development" branch only for my state repository. All environments (dev, qa, stage etc...) pulled from "release" branch of the state repository. When we need to develop new states or refactor existing states, we work on "development" branch, and after validating, this can be merged to "release" branch in the state repo just like you would release a software application/service etc....

(testing development branch states is easy in the salt environment as it's simply a parameter that can be toggled for the purpose of testing states before officially merging into release branch.)

So now we have a generic (parameterized) webserverX product state that can be implemented and/or deployed to different environments without the need to make copies of this state.

The parameters that are passed in when applying this state would come from environment delta configuration, team specific configuration (overrides) and such. These variable configurations would be defined in the Salt 'Pillar Tree". Using Jinja in salt states allows the states to be built generically. The jinja would "lookup" implementation specific config in the pillar tree when rendering the final salt state.

The "Pillar tree" in the Salt world is managed quite differently than the State tree since all your environment and configuration deltas are defined there. Thus the pillar tree repository usually follows a branching model that maps 1:1 with you "salt environments". Salt environments map 1:1 with you business environments. I.E dev, qa, stage prod would be business environments and thus you would have 4 salt environments. All enviroments would use the same state tree branch, but different dedicated pillar tree branches.

With this model, if you don't specify a config in you pillar branch, the defaut as defined in the state would be used.

Based on this design, you would never merge code between pillar branches as each branch acts as an isolated configurtion hierarchy for a dedicated environment.

There's much more depth to the architecture and how Saltstack pillar and state trees work than can be discussed in this overview. For example Salt internal uses a complex discovery mechnism to locate/lookup and find states, and config values. The design is highly scalable to not only supporting multiple environments from one Saltstack system, but if desired, a single Saltstack system could support many organizations with integrated or isolated state and pillar trees.

In summary, any setting, configuration values etc... for any product, app or service that might need to change between different environments or implementations, should be externalized from your state tree and defined in pillar tree.

Layered value overriding

Built in the ability for teams to override values (see extracting environment deltas). This is a very design specific topic. But in general any configuration management/delivery automation framework should support multiple levels of value overridding. For example Chef uses an elaborate attribute precedence hierarchy.

In the case of Salt, since Jinja is a core component of state and pillar, you would implement your overrides in how you choose to design your system.

Example of a 4 layer override: (implementation specific could refer to one teams deployment of productX vs another teams implementation of the same productX)

- product X salt state default settings
    - product X salt state environment specific settings (overrides default)
        - product X salt state implementation specific settings (could override both enviroment specific and/or default settings)
            - product X salt state runtime specific settings (special case override of any settings realized from the previous override evaluation)

Orchestration

Orchestration is interesting concept. It can be applied to anything. Whether you are building an enterprise wide delivery automation framework or running with a hodge-podge of technologies and solutions all disconnected from each other, orchestration comes into play. Disclaimer: I completely disagree with the latter approach with reasons being noted throughout this document.

However orchestration is always needed. In the latter example, your orchestration would simply be the glue to connect all these disparate systems in some soft of sequencial manor. A simple example is a script of script executions. A wrapper essentially.

In a more structured delivery automation framwork orchestration becomes a much more powerful and connected tool to the system. Your

orchestration can use the same configuration data as your instance/server, infrastructure and system configuration. Single source of record. Orchestration can be much more intelligent also if it's part of the overall delivery framework. Various phases of orchestration can "learn" about what's going on in various other phases of the orchestration and make logic decisions based on events. At this point your orchestration is smarter because it can make deterministic flow changes.

A good delivery framework typically would have multiple "orchestrations". For example, a build pipeline is an orchestration. The pipleine orchestrates many phases throughout the product delivery cycle.

code commit build test publish delploy test verify activate

promote (next env) deploy test verify activate

promote (next env) deploy test verify activate

This example pipe orchestrates the steps from code commit to activating the ready instance to accept traffic. Within the pipe there would be at least one additional orchestration, deploy.

The deploy phase of a CD pipeline is where your provisioning and delivery orchestration takes place. This document is mostly focused on that Orchestration.

Idempotency

Idempotency is a term used to describe the capability of applying configuration over and over without any risk of side effect or unwanted results.

Within provisioning and configuration management systems there should be a strategy to create idempotent systems. When writing automation to configuration a system, always think about what happens when you run the automation twice, solve any issues and then you will have idempotent systems in the environment.

Safe Configuration

This is a term I tend to use when designing enterprise wide automation and product delivery systems. What happens when a configuration is run/applied to an instance that it was not intended. The best answer, is nothing. This is 'Safe configuration'. It's related to idempotency in that you run something and no ill effects will result. However, the difference in Safe Configuration is that not only should nothing bad happen when run on the intended target machine, but the same if run on the non-intended target.

One method of acheiving this level of confidence in your automation framework is to use 'Role based' designs. All instances have at least one role. Role together with other metadata or instance labels allows very specific targeting of instances when applying configuration, but also provides a hook to be able to write configuration with some basic logic to determine that when run on a system, does it actually need to execute it's configuration on that system.

In the Saltstack world, we use grains for this and many other instance, cluster, system and environment identification.

Ephemeral systems

Aways try to build ephemeral system. This requires 100% automation with no human intervation. Kaos monkey and other methods of testing resiliency can be introduced once systems grow and become complex.

Starting at the basic machine level, everything should be "rinse and repeat". Standup a fully configure server or system, terminate one server and bring it back. The server should be idempotent and the system (cluster etc...) that the server is part of should all be able to be brought back to stablestate by automation.

Obviously, very good knowledge of the systems, software and technology at the architecture level is required! You have to know all you use cases including single or multi machine loss, zone or region loss etc...

Disaster recovery can become a very achievable thing when you have ephmeral systems throughout.

Service discovery

Incorporating a service discovery implementation within the automation framework goes a long way in establishing generic system deployment. Using service discovery to get and set dynamically created data and cloud metadata will allow an automation to be more generic and have less dependency on hard coded values.

Some configuration and deployment tools have some basic ability for service discovery, but mostly relying its own ecosystem.

However, when systems need to communicate with other systems or endpoints outside thier scope, it a good idea to implement a dedicated service discovery technology. Connecting communication channels between geographically and logically separated systems is one of the key advantages with using a dedicated technology.

Architecture

This section breifly summarizes the required systems needed

Build System

- pull source, build langauge (maven, sbt, gradle, nant, msbuild etc...), compile, test, publish to artifact management system, participates in a pipeline
- Jenkins, TFS etc...

Artifact management

- source of record for all build assets, remote 3rd party distro proxy, source of record for enterprise wide binary resources (jdk, etc...), particiates in a pipeline
- Artifactory, other....

Configuration Management

- defines the delivery software architecture, business workflow and operational lifecycle use cases for the organization in the form of configuration files within a system of automation.
- provides infrastructure as code, provisioning, deployment and delivery as code, mechanism for reusability, consistency in delivery across all facets and automation scalability
- Saltstack, Ansible, Chef etc....

Deployment execution

- might be internal to configuration management tool, provides user interface for execution of deployments on demand, might be the hook into starting the configuraiton management process
- receives hand-off from build process in a pileline, gets it's variable values from source repository or on the fly, tightly hooked into the configuration management system
- Built-into config management tools such as Saltstack, Ansible Tower, or other stand alone tools like Rundeck, Func etc...

Secret Management

Similar to a Service, system or configuration discovery tool, a secrets management system should be implemented along with the automation framework. The basic concepts are the same as service discovery as described in the next section, with the added requirement of tight encryted security.

Having a single source of record for password and other secrets provided benefits such as easy management when needed due to security breaches, a single place to perfrom auditing, secret retention and other policy based operations.

Systems offering an api such as Vault, Lastpass etc... and interfacing with them at all phases of automation (build, deployment, provisioning, operations) should be planned out.

Service Discovery

Service discovery is a crucial piece of any enterprise wide system. The ability for one system or service to find another dependent serivce or endpoint running on a different system, subnet or even different region is soemthing every organization will have to deal with. Having a centralized tool to management the 'discovery' of systems, services and configurations is an important thing to include in any design.

Service discovery is a general term, but really applies to the programmatic discovery of any of the above mentioned componenets. Discovery needs and requirements can be at the internal application software layer, the network layer between systems, or within a logical 'set' of systems, aka a cluster or some such.

Some configuration management technologies offer capabilities for discovery within its own ecosystem. Saltstack is a great example of that. However, the technology specific discovery may be limited and most useful when discovery is needed within and 'outside' the ecosystem.

Enterprise level service, system and configuration discovery should be implemented with a dedicated technology such as Consul, Zookeeper etc...

These dedicated discovery systems can span over global operations where not everything can be part of the configuration management ecosystem. But, they can and should interface with all systems including a configuration management system.

For example, in an automation framework (Salt, Ansible etc...) that used to provision, config and manage all your cloud instances, it is good practice to coelesce the metadata created for such systems and upload into the enterprise discovery system. So the two (internal and external) discovery systems work together.

Adoption