# Solution Engineering Guidance for Product Teams

Note: The following sections were compiled by the Architecture and Engineering Service (AES) as solution engineering principles and best practices that support Product Line Management and DevSecOps Product Management and address common technical architecture concerns. They are part of the enterprise technology standards that all IT products follow to ensure successful business outcomes.

Primary Resources:

* DevSecOps Handbook: [https://department-of-veterans-affairs.github.io/DevSecOps-Coms-Public/](https://department-of-veterans-affairs.github.io/DevOps-Coms-Public/)
* DevSecOps Enterprise Architecture (EA) Repository: <https://dvagov.sharepoint.com/sites/OITEPMOVAEA/devops/SitePages/Home.aspx>
* Technical Reference Model (TRM): <http://trm.oit.va.gov/>
* DevSecOps Release Process: [https://vaww.oit.va.gov/oit/DevSecOps/release-process/](https://vaww.oit.va.gov/oit/devops/release-process/)
* DevSecOps Product Management Lifecycle (DPML) ([Process Asset Library](https://vaww.oed.wss.va.gov/process/Lists/Processes/Item/displayifs.aspx?List=4ab19761%2De153%2D4db0%2Dad5a%2D6e9a42571488&ID=548&Web=88309b19%2D0f08%2D4d14%2D9f6d%2Dc2bbcf4a9f59)) ([Link](https://vaww.oed.wss.va.gov/process/Public/API/PP4/API/DisplayProcessOverview4.html?id=DPM))
* [DevSecOps Framework (GSA Guide)](https://vaww.vashare.oit.va.gov/sites/OneVaEa/DevOps/Shared%20Documents/DevSecOps%20References/DevSecOps%20Guide%20-%20Tech%20at%20GSA%20(Reference%20for%20VA).pdf)
* Digital Services Handbook: <https://department-of-veterans-affairs.github.io/va-digital-service-handbook/>
* Agile Center of Excellence (ACOE): <https://dvagov.sharepoint.com/sites/OITACOEPortal/Pages/home.aspx>
* VA Enterprise Cloud (VAEC) (includes Azure and AWS Welcome Packages): <https://dvagov.sharepoint.com/sites/OITEPMOECSO/SitePages/VA-Enterprise-Cloud-VAEC.aspx>
* VA Cloud Software Catalog: <https://www.oit.va.gov/services/cloud-software/catalog>
* GitHub Handbook: <https://department-of-veterans-affairs.github.io/github-handbook/>

Infrastructure as Code (IaC): Improved Code Reuse, Faster Deployments, Lower Cost

Primer from VA industry partner - <https://www.ibm.com/cloud/learn/infrastructure-as-code>

Agile Architecture Best Practices - <https://tanzu.vmware.com/content/blog/agile-architecture>

Reusable templates provided for the VAEC: <https://github.com/Azure/azure-quickstart-templates>.

Also useful to solution architecture are the following template references:

* AWS CloudFormation: <https://aws.amazon.com/cloudformation/resources/templates/>
* Pre-approved Terraform template starters for [AWS](https://www.terraform.io/docs/providers/aws/index.html) and [Azure](https://docs.microsoft.com/en-us/azure/terraform/terraform-overview)
* Hashicorp Packer <https://www.packer.io/docs/templates/index.html>
* API design: <https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design> (REST)
* [Red Hat OpenShift Templates Developer Guide](http://v1.uncontained.io/playbooks/fundamentals/template_development_guide.html)
* [IBM Cloud Paks](https://www.ibm.com/cloud/learn/infrastructure-as-code) for Apps and Multicloud Management
* [Kubernetes versus OpenShift – guidance on using pure Kubernetes versus OpenShift](https://hackernoon.com/kubernetes-vs-openshift-a-detailed-comparison-7r3z53zlv)
* [Kubernetes Workloads in the Serverless Era: Architecture, Platforms, and Trends](https://vaww.vashare.oit.va.gov/sites/OneVaEa/DevOps/Shared%20Documents/Technical%20Architecture%20Standards%20and%20Guidance/Kubernetes%20Workloads%20in%20the%20Serverless%20Era_%20Architecture,%20Platforms,%20and%20Trends.pdf)

Solution engineering and production practices:

* Periodic tests of in-scope components in a staging environment in a continuous deployment pipeline
* Upstream components are known to be load-tested up to max foreseeable pressure
* Automated pen-testing in a staging environment as part of continuous deployment
* Automated vuln-scanning in production environment that is fed with newly-discovered vulns
* Automated environment builds and software environment standardization requirements

# Key Guiding Principles

* Stay abreast with emerging technologies including DevSecOps toolchains such as those provided by the DevSecOps Team (DOTS) under the CTO ([DevOpsTeam@va.gov](mailto:DevOpsTeam@va.gov))
* Alignment to [Play 9](https://playbook.cio.gov/#play9) and [Play 10](https://playbook.cio.gov/#play10) in the Digital Services Playbook
* Digital Standards: <https://department-of-veterans-affairs.github.io/va-digital-service-handbook/digital-standards>
* 12 Factor App – Separate environments, parity between pre-production and production
  + [X. Dev/prod parity](http://12factor.net/dev-prod-parity)
  + [V. Build, release, run](http://12factor.net/build-release-run)
* VAEC Architecture (DevTest, N-Prod, and Prod Environments)
  + [VAEC-AWS Welcome Kit](https://vaec-confluence.ec.va.gov/display/VK/AWS+Welcome+Packet)
  + [VAEC-Azure Welcome Kit](https://vaec-confluence.ec.va.gov/display/VK/Azure+Welcome+Packet)
* Azure Architecture: <https://docs.microsoft.com/en-us/azure/architecture/>
* AWS Well-Architected Framework: <https://wa.aws.amazon.com/index.en.html>
* Application Continuum: <https://www.appcontinuum.io/>
* Agile Architecture: <https://tanzu.vmware.com/content/blog/agile-architecture>

The [VA Digital Delivery Guide](https://department-of-veterans-affairs.github.io/va-digital-service-handbook/delivery/index.html) is an excellent resource that walks through the phases of product development, including (1) research and discovery, (2) prototype, (3) build and test, and (4) learn and improve, including specific questions it will be helpful to ask and answer, as well as activities it will be valuable to complete.  Although it is at the discretion of the product owner and not mandatory to follow every step in the VA Digital Delivery Guide, the guide can be valuable to reference and learn from.

Best Practices and Guidance (from Gov.UK): When considering technical architecture, choice of programming languages, development toolchain and other technology choices, product teams should:

* Using appropriate [tools and technologies](https://www.gov.uk/service-manual/technology/choosing-technology-an-introduction) to create and operate a good service in a cost effective way - for example, by automating things where possible
* Making [good decisions](https://www.gov.uk/service-manual/technology/choosing-technology-an-introduction) about what technology to build and what to buy
* Understanding total cost of ownership of the technology and preserve the ability to make different choices in future - for example, reducing the chances of getting locked into contracts for specific tools and suppliers by [using open standards](https://www.gov.uk/service-manual/technology/working-with-open-standards)
* [Managing any legacy technology](https://gds.blog.gov.uk/2014/07/08/making-prison-visits-easier-to-book/) the service integrates with or depends on

## ***Change Management Guidance:***

The basic steps associated with the Change Management process for VA are as follows:

1. Initiate Change
2. Analyze/Plan Change
3. Approve Change
4. Fix/Develop Change
5. Implement Change
6. Validate Change

CI and CD form the practice of frequently building and testing each change made to the code baseline automatically and as early as possible. CI is accomplished via the automation of functional and security-based testing. By automating testing efforts using tools including security scans, users can ensure that the most important features of applications are both working and secure regardless of the changes made by the development team. This enables new features to be confidently implemented in a secure manner while ensuring updates can be deployed quickly. Developers use CI workflows to discover coding flaws early during the project or sprint saving time and preventing deployment delays of new features.

This workflow is unchanged for infrastructure builds, application builds, or both. Based on specific VA project requirements, an artifact may be rebuilt after being merged into the Release Branch. Neither method will reduce feature parity for all targeted environments (Dev/Test/Prod…). This CD mechanism requires environment-specific configuration to be parametrized and removed from the binary artifact. These parameters will be stored in GitHub thus ensuring each environment is under strict CM control and adhering to established VA process/procedures.

## ***Build Automation Guidance:***

<https://docs.microsoft.com/en-us/azure/jenkins/overview>

A screenshot of a cell phone

Description automatically generated

Jenkins is the most commonly used automation server for creating builds and running tests against these builds. Jenkins is recommended for all VA teams. Other VA teams are using Circle CI and Travis CI, which can provide similar functionality and may meet a team’s needs as well.

Jenkins Documentation (approved in TRM): <https://jenkins.io/doc/> and <https://plugins.jenkins.io/>. The Jenkins job will turn green when a deploy has completed successfully, the job will turn green on the console. A failed job will turn red.

Different continuous integration tools for different challenges:

<https://hackernoon.com/continuous-integration-circleci-vs-travis-ci-vs-jenkins-41a1c2bd95f5>

Using CircleCI or Travis CI? Check out examples from Caseflow product: [CircleCI](https://circleci.com/gh/department-of-veterans-affairs/caseflow), [Travis CI](https://travis-ci.org/github/department-of-veterans-affairs/caseflow-commons)

Cloud-native Technology Dependencies:

* Immutable infrastructure and Infrastructure as Code (IaC): Supports reusable code, faster deployments, and lower costs
  + Terraform is commonly used for building infrastructure and Ansible is commonly used for application deployments and updates. Unlike Ansible, Terraform does not offer configuration management capabilities, but it works together with configuration management tools (e.g., CloudFormation) to automatically provision infrastructure in the state described by configuration files and to automatically change update provisioning when necessary in response to configuration changes.
    - [Terraform Modules Guidance](https://www.terraform.io/docs/modules/index.html)
  + [Example: Jenkins with Ansible](https://www.redhat.com/en/blog/integrating-ansible-jenkins-cicd-process)
  + Ansible uses YAML (modules written in Python) while Terraform uses [HCL](https://www.linode.com/docs/applications/configuration-management/introduction-to-hcl/)
  + IBM overview of IaC: <https://www.ibm.com/cloud/learn/infrastructure-as-code>
  + [Comparing configuration management tools (reflects TRM status)](https://en.wikipedia.org/wiki/Comparison_of_open-source_configuration_management_software)
* Reach out to the VAEC for the Azure EDE Lab (DMAG) or DevTest Lab or the Emerging Technologies Innovation Lab (POC: Angela Gant-Curtis)
  + The ETIL includes [Synthea](https://www.mitre.org/publications/project-stories/synthetic-patient-records-help-deliver-real-health-outcomes) for synthetic data in a dev/test environment before using live data in production: <https://github.com/synthetichealth/synthea>
* VAEC new application development policy, follow standard dev workflows
  + <https://github.com/department-of-veterans-affairs/va.gov-vfs-teams/blob/master/DeveloperDocs/development-workflow.md>
  + Deployment Process: <https://github.com/department-of-veterans-affairs/va.gov-vfs-teams/blob/master/DeveloperDocs/deployment.md>
* Linters and security vulnerability scans should also be run automatically at the same time.  If any of the tests or scans fail, the changes should not be allowed to be merged into the master branch. Automated accessibility tests should exist for any user-facing functionality.
* Kubernetes versus OpenShift – cost-benefit analysis for legacy applications versus greenfield application development: <https://www.youtube.com/watch?v=cTPFwXsM2po>
* TRM compliance through VMs and containers that run on the VA network
  + [Container security policy follows NIST standards](https://vaww.vashare.oit.va.gov/sites/OneVaEa/DevOps/Shared%20Documents/DevSecOps%20References/NIST.SP.800-190.pdf)
  + Processes in a container should not run as root or assume that they are root. Instead, create a user in your Dockerfile with a known UID and GID, and run process as this user.

DevSecOps Release Process Mandatory Pre-requisites:

[https://vaww.oit.va.gov/oit/DevSecOps/release-process/mandatory-prerequisites/](https://vaww.oit.va.gov/oit/devops/release-process/mandatory-prerequisites/)

## ***Development Guidance:***

**Provide developers with production-like environments.** If development and test environments don't match the production environment, it is hard to test and diagnose problems. Therefore, keep development and test environments as close to the production environment as possible. Make sure that test data is consistent with the data used in production, even if it's sample data and not real production data (for privacy or compliance reasons). Plan to generate and anonymize sample test data.

**Ensure that all authorized team members can provision infrastructure and deploy the application.** Setting up production-like resources and deploying the application should not involve complicated manual tasks or detailed technical knowledge of the system. Anyone with the right permissions should be able to create or deploy production-like resources without going to the operations team.

This recommendation doesn't imply that anyone can push live updates to the production deployment. It's about reducing friction for the development and QA teams to create production-like environments.

**Instrument the application for insight.** To understand the health of your application, you need to know how it's performing and whether it's experiencing any errors or problems. Always include instrumentation as a design requirement, and build the instrumentation into the application from the start. Instrumentation must include event logging for root cause analysis, but also telemetry and metrics to monitor the overall health and usage of the application.

**Track your technical debt.** In many projects, release schedules can get prioritized over code quality to one degree or another. Always keep track when this occurs. Document any shortcuts or other nonoptimal implementations, and schedule time in the future to revisit these issues.

**Consider pushing updates directly to production.** To reduce the overall release cycle time, consider pushing properly tested code commits directly to production. Use [feature toggles](https://www.martinfowler.com/articles/feature-toggles.html) to control which features are enabled. This allows you to move from development to release quickly, using the toggles to enable or disable features. Toggles are also useful when performing tests such as [canary releases](https://martinfowler.com/bliki/CanaryRelease.html), where a particular feature is deployed to a subset of the production environment.

## ***Testing Guidance:***

**Playbook for Software Testing and 508 (EPMD):** <https://vaww.oed.portal.va.gov/sites/enterprisetestingservice/sandbox/DemoPlaybook/Pages/default.aspx>

**Automate testing.** Manually testing software is tedious and susceptible to error. Automate common testing tasks and integrate the tests into your build processes. Automated testing ensures consistent test coverage and reproducibility. Integrated UI tests should also be performed by an automated tool. Azure offers development and test resources that can help you configure and execute testing. For more information, see [Development and test](https://azure.microsoft.com/solutions/dev-test/).

Today, developers write automated scripts that can verify thousands of scenarios in minutes and then deploy updated code into production environments multiple times a day. They use automated performance tests which simulate surges in traffic to identify performance bottlenecks. While manual tests and quality assurance are still necessary, automated tests provide consistent and reliable protection against unintentional regressions, and make it possible for developers to confidently release frequent updates to the service.

Checklist

1. Create automated tests that verify all user-facing functionality
2. Create unit and integration tests to verify modules and components
3. Run tests automatically as part of the build process
4. Perform deployments automatically with deployment scripts, continuous delivery services, or similar techniques
5. Conduct load and performance tests at regular intervals, including before public launch

Key Questions

1. What percentage of the code base is covered by automated tests?
2. How long does it take to build, test, and deploy a typical bug fix?
3. How long does it take to build, test, and deploy a new feature into production?
4. How frequently are builds created?
5. What test tools are used?
6. Which deployment automation or continuous integration tools are used?
7. What is the estimated maximum number of concurrent users who will want to use the system?
8. How many simultaneous users could the system handle, according to the most recent capacity test?
9. How does the service perform when you exceed the expected target usage volume? Does it degrade gracefully or catastrophically?
10. What is your scaling strategy when demand increases suddenly?

**Test for failures.** If a system can't connect to a service, how does it respond? Can it recover once the service is available again? Make fault injection testing a standard part of review on test and staging environments. When your test process and practices are mature, consider running these tests in production.

**Test in production.** The release process doesn't end with deployment to production. Have tests in place to ensure that deployed code works as expected. For deployments that are infrequently updated, schedule production testing as a regular part of maintenance.

**Automate performance testing to identify performance issues early.** The impact of a serious performance issue can be as severe as a bug in the code. While automated functional tests can prevent application bugs, they might not detect performance problems. Define acceptable performance goals for metrics like latency, load times, and resource usage. Include automated performance tests in your release pipeline, to make sure the application meets those goals.

**Perform capacity testing.** An application might work fine under test conditions, and then have problems in production due to scale or resource limitations. Always define the maximum expected capacity and usage limits. Test to make sure the application can handle those limits, but also test what happens when those limits are exceeded. Capacity testing should be performed at regular intervals.

After the initial release, you should run performance and capacity tests whenever updates are made to production code. Use historical data to fine-tune tests and to determine what types of tests need to be performed.

**Perform automated security penetration testing.** Ensuring your application is secure is as important as testing any other functionality. Make automated penetration testing a standard part of the build and deployment process. Schedule regular security tests and vulnerability scanning on deployed applications, monitoring for open ports, endpoints, and attacks. Automated testing does not remove the need for in-depth security reviews at regular intervals.

**Perform automated business continuity testing.** Develop tests for large-scale business continuity, including backup recovery and failover. Set up automated processes to perform these tests regularly.

Leverage automation and tooling to help do the following:

* Leverage user definable patterns and code templates to automate repetitive coding tasks
* Increase productivity of developers by automatically generating code and database schemas
* Roundtrip engineering for data models and code
* Automatic or on demand synchronization of models and code
* ServiceNow YourIT ticket for requesting 508 testing

## ***Test Environments Guidance:***

Continuously test the end-to-end service in an environment identical to that of the live version. Doing this helps you find problems before you launch a live version, which means you’re more likely to release something that works for your users.

<https://department-of-veterans-affairs.github.io/va-digital-service-handbook/digital-standards>

At a minimum,

* Design your service so it accommodates the expected number of users and can support more users if demand increases
* Separate content, design, and functionality so updates can be made independently of each other
* Follow recommended best practices for coding in your chosen tools
* Test the service in the staging environment using the required automated testing tools
* Follow the process for frequently monitoring and testing your service, even when changes are not being made
* Follow the process for managing failures (bugs, outages) and how to notify users
* Follow the process for data storage and recovery in case of data loss
* Document how your service was built and how to maintain it, and keep the documentation up-to-date

Once the code has been reviewed and approved, the feature branch should be merged into the master branch. This act should automatically begin a process in which all tests, including long running tests that were not run when the pull request was created because of the length of time they take, such as accessibility scans run to verify 508 compliance or code security scans, should be run on the master branch after each merge to test for regressions.

Once all tests pass, the code should be automatically deployed to a non-production environment (often known as “development” or “QA”). If the tests do not pass on the master branch (or other main development branch), the team should return to Step 1 to create a ticket and quickly address the problem. Tests should rarely fail on the master branch and when they do, it should be remediated as quickly as possible. To keep velocity high, the master branch must not be left in a state in which tests are failing, since all members of the team will be referencing the validity of their specific changes against the master branch. This non-production environment gives the team, stakeholders, and product owner the ability to test the updated functionality. Usability testing may also be performed on this non-production environment to test for any bugs or usability issues prior to the code being deployed to production.

**Testing in the Cloud**: In the cloud, you can create a production-scale test environment on demand, complete your testing, and then decommission the resources. Because you only pay for the test environment when it's running, you can simulate your live environment for a fraction of the cost of testing on premises.

* VAEC will provide development and test environments for application project teams
* When projects start the VIPR process, ECSO can open areas that the team can use to start to test and develop in a cloud environment. This will ensure that project teams are better prepared to support an application’s migration to the cloud
  + Enabling quicker access to develop your product on the VAEC
  + Providing access while credentials are being established
* Have developers build unit tests toward 100% coverage of the code base
* Ensure that unit tests are 70% of the overall testing in duration, number, and scope
* It is a best practice to send a small percentage of traffic to the new version of the code and measure for unusual activity or responses, before ramping up traffic to the new version, eventually getting to 100% of traffic on the new version and being able to automatically destroy the previous environment.
* If necessary, it should be possible to quickly “roll back” any production deployment either by re-deploying the previous version, or falling back to the previous copy of “production” which will remain deployed for some period (i.e. an hour).
* Except in rare cases where downtime is anticipated overtime is planned, it is a best practice to avoid issuing major releases on Fridays, weekends, days before holidays, or other periods where personnel are likely to be difficult to reach in case issues arise.

**Container-based Sandboxes Best Practice**: An example of this can be found in the Docker based sandbox for smart apps: <https://github.com/smart-on-fhir/smart-dev-sandbox>

The SMART Dev Sandbox is an open source, Docker based version of the SMART Sandbox that can be installed locally on your machine to facilitate offline development and the use of custom data sets. Please note that it is not designed for clinical use and should not be used to store or access patient medical data.

# CI/CD Best Practices and Guidance

Source: Amazon Web Services (AWS)

DO:

* Treat your infrastructure as code
  + Use version control for your infrastructure code.
* Make use of bug tracking/ticketing systems.
  + Have peers review changes before applying them.
  + Establish infrastructure code patterns/designs.
  + Test infrastructure changes like code changes.
* Put developers into integrated teams of no more than 12 self-sustaining groups.
* Have all developers commit code to the main trunk frequently, with no long-running feature branches.
* Consistently adopt a build system such as Maven or Gradle across your organization and standardize builds.
* Have developers build unit tests toward 100% coverage of the code base.
* Ensure that unit tests are 70% of the overall testing in duration, number, and scope.
* Ensure that unit tests are up-to-date and not neglected. Unit test failures should be fixed, not bypassed.
* Treat your continuous delivery configuration as code.
* Establish role-based security controls (that is, who can do what and when).
  + Monitor/track every resource possible.
  + Alert on services, availability, and response times.
  + Capture, learn, and improve.
  + Share access with everyone on the team.
  + Plan metrics and monitoring into the lifecycle.
* Keep and track standard metrics.
  + Number of builds.
  + Number of deployments.
  + Average time for changes to reach production.
  + Average time from first pipeline stage to each stage.
  + Number of changes reaching production.
  + Average build time.
* Use multiple distinct pipelines for each branch and team.

DON’T:

* Have long-running branches with large complicated merges.
* Eliminate manual tests.
* Eliminate manual approval processes, gates, code reviews, and security reviews.

Build automation is essential to the CI process. When setting up build automation, the first task is to choose the right build tool. There are many build tools, such as Ant, Maven, and Gradle for Java; Make for C/C++; Grunt for JavaScript; and Rake for Ruby. The build tool that will work best for you will depend on the programming language of your project and the skill set of your team. After you choose the build tool, all the dependencies need to be clearly defined in the build scripts, along with the build steps. It’s also a best practice to version the final build artifacts, which makes it easier to deploy and to keep track of issues.

In the build stage, the build tools will take as input any change to the source code repository, build the software, and run the following types of tests:

**Unit Tests –** Tests a specific section of code to ensure the code does what it is expected to do. The unit test is performed by software developers during the development phase. At this stage, a static code analysis, data flow analysis, code coverage, and other software verification processes can be applied.

**Static Code Analysis –** This test is performed without actually executing the application after the build and unit testing. This analysis can help to find coding errors and security holes, and it also can ensure conformance to coding guidelines (SONAR).

Staging

In the staging phase, full environments are created that mirror the eventual production environment. The following tests are performed:

**Integration Testing** – Verifies the interfaces between components against software design. Integration testing is an iterative process and facilitates building robust interfaces and system integrity.

**Component Testing** – Tests message passing between various components and their outcomes. A key goal of this testing could be idempotency in component testing. Tests can include extremely large data volumes, or edge situations and abnormal inputs.

**System Testing** – Tests the system end-to-end and verifies if the software satisfies the business requirement. This might include testing the UI, API, backend logic, and end state.

**Performance Testing –** Determines the responsiveness and stability of a system as it performs under a particular workload. Performance testing also is used to investigate, measure, validate, or verify other quality attributes of the system, such as scalability, reliability, and resource usage. Types of performance tests might include load tests, stress tests, and spike tests. Performance tests are used for benchmarking against predefined criteria.

**Compliance Testing –** Checks whether the code change complies with the requirements of a nonfunctional specification and/or regulations. It determines if you are implementing and meeting the defined standards.

**User Acceptance Testing –** Validates the end-to-end business flow. This testing is executed by an end user in a staging environment and confirms whether the system meets the requirements of the requirement specification. Typically, customers employ alpha and beta testing methodologies at this stage.

# Java Gitflow Pipeline Example

The CI/CD pipeline models the [Gitflow development model](https://datasift.github.io/gitflow/IntroducingGitFlow.html). New release is done on a 'development' branch with developers contributing using PRs against this branch. When the release is ready, a tag is made and promoted to staging. Once validated in staging and deemed ready for production, the tag is merged into the master branch which triggers a **production** deployment.

BIP Example:

<https://github.com/department-of-veterans-affairs/bip-framework>

<https://nexus.dev.bip.va.gov/> (Nexus artifact repository example from BIP)

<https://github.com/jenkinsci/docker/blob/master/README.md>

1. A test log, such as the output from Jenkins, should be permanently stored to validate that the tests, scans, and linters all passed. Additionally, a reference, such as a GitHub Release, should be created to the exact code that was deployed to production.  This documentation allows auditability of what ran before code got to production and what code was in production at a specific time.
2. In short, Archetype is a Maven project templating toolkit. An archetype is defined as an original pattern or model from which all other things of the same kind are made. The name fits as we are trying to provide a system that provides a consistent means of generating Maven projects. Archetype will help authors create Maven project templates for users, and provides users with the means to generate parameterized versions of those project templates.
3. An artifact, such as a GitHub release, should be created to automatically show the differences in version control from the previous deployment. This allows tracking of what changed between each release (for both compliance and debugging if issues arise) without adding additional burden of humans having to document this manually.
   * **Sonar Code Quality Analysis**  
     Sonar code findings added to PR as comments. Fail build if [Sonar Quality Gate](https://github.ec.va.gov/EPMO/bip-jenkins-lib/blob/master/docs/common/sonar.md) is not passed.
   * **Fortify Security Code Analysis**  
     Fortify security scans are run on each commit and the finding evaluated against quality standards. Fail build if [Fortify Quality Gate](https://github.ec.va.gov/EPMO/bip-jenkins-lib/blob/master/docs/common/fortify.md) is not passed.

# TDD and Ruby, and Environment Normalization Guidance

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | Projects using this technology must employ a Test-Driven development model and allow sufficient time to the functional testing phase to ensure adequate testing of dynamic typing. Best Practice: You can’t begin to TDD until you know the software architecture that you’ll TDD within. The software architecture flows from the desired user experience. | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |

<https://medium.com/@khandelwal12nidhi/ci-cd-of-ruby-on-rails-using-jenkins-e7fb47a14aae>

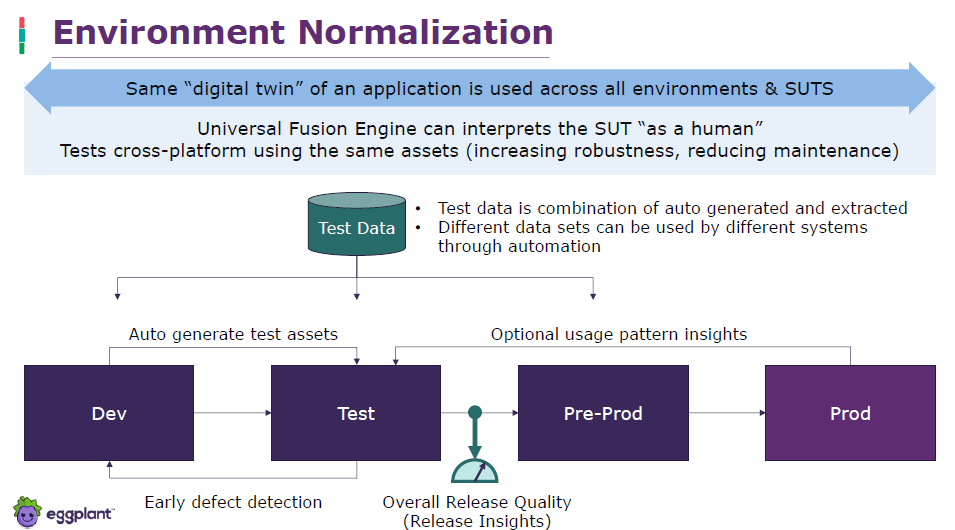
<https://rspec.info/>

<https://www.codewithjason.com/difference-rspec-capybara-cucumber/>

* A good practice is to write tests prior to implementing the feature; the test should fail, and then pass upon completion of the feature.

Eggplant also works with Ruby and Jenkins Plugin:

|  |
| --- |
| Eggplant Automation Intelligence (AI) works together with other Eggplant solutions to meet load and capacity testing needs. This is a mature technology that could improve productivity for employees who utilize other Eggplant solutions. Additionally, at the time of writing, there are no security vulnerabilities associated with this technology. |
|  | |
| |  | | --- | | Jenkins Plugin is a plugin application designed for automated, performance, and user interface (UI) testing with Jenkins. Users may be able to host multiple environments for building and running tests. This technology can help with task distribution, which may help free up resources in build systems. Additionally, users may have the ability to better enact deployments into secured environments. | |  | | | |



## ***Environment Variables Guidance:***

Environment variables are pairs of key-and-value that can be used in the pipelines instead of manually updating the same values. They allow developers to save time looking up and entering access details and eliminate the risk of providing sensitive details in their scripts.

ENV VARs are used to parameterize build scripts. Setting up variables in a pipeline will cause the same build action to work differently depending on the pipeline. See Appendix on 12-factor app “store config in the environment.” Chef, for example, makes use of configuration setting files known as `cookbooks` which may be stored in a local directory or remotely on a Uniform Resource Locator (URL). Cookbooks are comprised of user defined `recipes` that describe resources and their states including software packages, services or files. In Ansible/Terraform, they are captured in Modules.

Jenkinsfile developer guide: <https://jenkins.io/doc/book/pipeline/jenkinsfile/#using-environment-variables>

Use of sample templates or custom ones –

<https://aws.amazon.com/cloudformation/aws-cloudformation-templates/>

<https://azure.microsoft.com/en-us/resources/templates/>

Specific guidance on how to use the EDE is as follows:

Azure EDE lab: <https://vaww.oed.portal.va.gov/pm/ede/default.aspx>

<https://vaww.oed.portal.va.gov/pm/EDE/Documents/Forms/EDE%20Technical.aspx> (older technical detail)

GovCloud EDE lab: set up an automated method to support provisioning standard sandbox/dev environments (unconnected to the VA network) in AWS “on demand”

## ***Runbooks Guidance:***

Installation Runbook for the Infrastructure Configurations and Software Installations. The Government expects the Runbooks contain procedures that need documented which include overview, how to build, how to deploy software, how build a server, RAM/Disk requirements, OS version and configuration, what packages to install, etc. If the Runbook is automated with a configuration management tool like cfengine/puppet/chef, please document this information in the runbook. The Contractor shall ensure reference documentation is part of the Runbook, Network Diagrams, Login Credentials, and configuration information. The Runbooks shall contain diagrams, tables, and procedures for the building/provisioning the Cloud Environments, Enterprise Monitoring Solution, user provisioning based upon the design documentation provided by the engineering team.

Note about Ansible for Windows:

Ansible can bring automation to a mixed operating system environment and provides an efficient way to get to an infrastructure-as-code (IaC) state without burdening your administrators. Ansible is not a replacement for System Center Configuration Manager ([SCCM](https://www.microsoft.com/en-us/cloud-platform/system-center-configuration-manager)) or [Chocolatey](https://chocolatey.org/); it's a supplemental tool that allows you to automate the services your software provides.