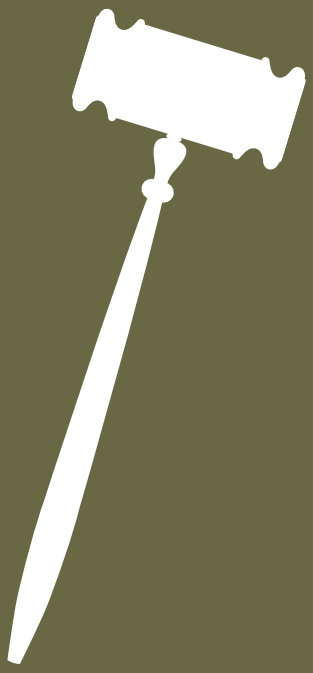
DevOps Automated Governance Reference Architecture



Attestation of the
Integrity of Assets
in the Delivery Pipeline

IT REVOLUTION
DEVOPS ENTERPRISE FORUM
2019

25 NW 23rd Pl
Suite 6314
Portland, OR 97210

DevOps Automated Governance Reference Architecture

Produced in the United States of America

Cover design and interior by Devon Smith

For further information about IT Revolution, these and other publications, special discounts for bulk book purchases, or for information on booking authors for an event, please visit our website at ITRevolution.com.

The 2019 DevOps Enterprise Forum was sponsored by XebiaLabs.

Preface

In May of this year, the fifth annual DevOps Enterprise Forum was held in Portland, Oregon. As always, industry leaders and experts came together to discuss the issues at the forefront of the DevOps Enterprise community and to put together guidance to help us overcome and move through those obstacles.

This year, the group took a deeper dive into issues we had just begun to unpack in previous years, providing step-by-step guidance on how to implement a move from project to product and how to make DevOps work in large-scale, cyber-physical systems, and even a more detailed look at conducting Dojos in any organization. We also approached cultural and process changes like breaking through old change-management processes and debunking the myth of the full-stack engineer. And of course, we dived into the continuing question around security in automated pipelines.

As always, this year's topics strive to address the issues, concerns, and obstacles that are the most relevant to modern IT organizations across all industries. Afterall, every organization is a digital organization.

This year's Forum papers (along with our archive of papers from years past) are an essential asset for any organization's library, fostering the continual learning that is essential to the success of a DevOps transformation and winning in the marketplace.

A special thanks goes to Jeff Gallimore, our co-host and partner and co-founder at Excella, for helping create a structure for the two days and the weeks that followed to help everyone stay focused and productive. Additional thanks goes to this year's Forum sponsor, XebiaLabs. And most importantly a huge thank you to this year's Forum participants, who contribute their valuable time and expertise and always go above and beyond to put together these resources for the entire community to share and learn from.

Please read, share, and learn, and you will help guide yourself and your organization to success!

—Gene Kim
June 2019
Portland, Oregon

Collaborators

Michael Nygard
VP Enterprise Architecture, Sabre GLBL

Tapabrata Pal
Senior Director and Senior Distinguished Engineer, CapitalOne

Stephen Magill
Principal Scientist and Software Analysis, Galois

Sam Guckenheimer
Product Owner of Microsoft Visual Studio Cloud Services, Microsoft

John Willis
Founder, Botchagalupe Technologies

John Rzeszotarski
Senior Vice President of Technology Infrastructure, PNC

Dwayne Holmes
Senior Director of IT Technology, Marriott International

Courtney Kissler
VP, Global Technology, NIKE, Inc.

Dan Beauregard
VP, Cloud & DevOps Evangelist, XebiaLabs

Collette Tauscher
Director of Strategy, Digital, NIKE, Inc.

Introduction

As organizations adopt DevOps practices, they develop increased productivity within their software development teams, faster releases of digital products, and improved customer experiences. But as the rate of delivery increases, it becomes more difficult for security and compliance to keep up without getting in the way. So, how can you ensure that all aspects of your deployment pipeline are protected as delivery velocity dramatically increases?

The "shift-left" practice in DevOps helps organizations improve quality and security by moving testing earlier in the release process. As more and more DevOps practices are automated, it becomes harder to capture the data required to ensure all security and compliance concerns are met. Organizations need an automated way to track governance throughout the entire software delivery process so they can attest to the integrity of all assets and to the security of all running applications.

This paper is intended to guide organizations on implementing an automated process for tracking governance throughout the deployment pipeline by providing a reference architecture to help guide organizations on how to design and implement automated governance throughout the delivery pipeline. A sample use case is also provided to further enforce these best practices.

Goals

Our initial focus was to design a model flexible enough that it could easily be extended and adopted by organizations struggling to maintain compliance and audit controls as their software delivery speed increased. We wanted to create a reference architecture that enables an organization to create trust within the process of delivering software and services. As organizations further automate the continuous

delivery of software and services, they also need to ensure there are common validations and trust mechanisms throughout the process. Ultimately, a DevOps automated governance process can give organizations the assurance that the delivery of their software and services are trusted.

The paper is organized into the following sections:

- Definitions and assumptions
- Reference architecture
- Recording attestations
- Summary and next steps

Disclaimer

The specific purpose of this paper is to introduce a reference architecture for the purposes of automated governance. This paper represents the combined knowledge of the authors' experiences in what should make a good start to create a DevOps automated governance process. We believe that this reference architecture will be a useful starting point for many organizations; however, it is probably not efficient for enterprise scale. It's the authors' expectation and hope that the DevOps community will engage in ongoing improvement through rigorous validation and continuous feedback. Furthermore, this paper is not intended to cover a comprehensive discussion regarding policy in the delivery pipeline. Though some of the control points described in this paper reference policy, we leave the instrumentation of that policy up to the reader.

Definitions and Assumptions

The following definitions and assumptions are used as the basis for this paper and can be modified to comply with each organization's preferred terminology.

Governance, Risk, and Compliance

The complexity of most modern organizations makes it very difficult for one team or even one person to understand a singular, comprehensive view of organizational

governance, risk, and compliance. As a result, most organizations typically apply governance, risk, and compliance (GRC) in an uncoordinated and nonaligned fashion.

In *Measuring and Managing Information Risk: A FAIR Approach*, Jack Freund and Jack Jones describe a more specific overview of GRC as follows:

- **Governance:** "Ultimately, leadership is expected to cost-effectively govern the organization's risk landscape. Accomplishing this requires setting and communicating expectations, overseeing and facilitating the achievement and maintenance of those expectations, and managing conditions that don't align with their expectations. GRC solutions are supposed to assist with this by providing a way to report where these expectations are and are not being met, within a meaningful business context."[1]
- **Risk:** "This objective is all about making better-informed risk decisions, which boils down to three things: (1) identifying 'risks,' (2) effectively rating and prioritizing 'risks,' and (3) making decisions about how to mitigate 'risks' that are significant enough to warrant mitigation."[2]
- **Compliance:** "Of the three objectives, compliance management is the simplest—at least on the surface. On the surface, compliance is simply a matter of identifying the relevant expectations (e.g., requirements defined by Basel, Payment Card Industry (PCI), SOX, etc.), documenting and reporting on how the organization is (or is not) complying with those expectations, and tracking and reporting on activities to close any gaps."[3]

When thinking about a "DevOps Automated Governance" model, we need to look specifically at *risk* as a probability or threat of damage, loss, or other negative occurrence that can be caused by external or internal vulnerabilities, and that may be avoided through preemptive action. Risk may manifest as direct or indirect losses. We then apply *governance* to understand and control risk.

Governance uses *controls* to mitigate specific risks. Controls can be classified as:

- **Detective:** the control indicates when a risk has already manifested.
- **Corrective:** the control repairs the process to compensate for a risk.
- **Preventive:** the control makes the risk less likely to manifest.

A governance process applies a variety of controls to diminish, mitigate, or respond to various risks. The governance process must both collect evidence that the controls are applied correctly and convey the results of the controls. Output from the controls can be collected in the form of *attestations*, which are witnessed declarations of evidence. Of course, attestations must be recorded in a tamper-resistant mechanism.

A key category of risk is the potential loss due to noncompliance with regulations or laws. Internal and external auditors attempt to verify compliance or detect noncompliance with these laws and regulations. Some examples of questions to be answered by various kinds of audits include:

- Are the company's financial records accurate?
- Does this piece of automotive software meet safety standards?
- Are the reported results of a drug trial what patients actually experienced?

Companies must periodically present evidence to these auditors that demonstrates compliance. A lack of evidence will be interpreted as noncompliance. Organizations therefore create governance mechanisms to ensure that the necessary evidence exists.

Every audit, for every purpose, requires data that is produced, managed, and reported by software. That means every audit eventually confronts the question of the integrity of that software.

Common Terminology

Throughout this paper, we will be using specific terms to describe certain aspects related to the DevOps automated governance reference architecture. Since the IT industry tends to overload a lot of this common terminology, we decided to use a common set of definitions for the purposes of this paper, listed below:

- **Delivery Pipeline:** This is the set of stages that describes how software will flow from post ideation to final production delivery. We use this phrase for all references related to industry terms, including but not limited to ARO, continuous delivery and release automation, continuous integration and continuous delivery, orchestration pipeline, release coordination, release management,

and software development life cycle. We also decided to use a common set of stages to describe the Delivery Pipeline as seen in Figure 1.

- **Artifact:** An artifact is a deployable component of an application. In this paper, we define an artifact as either an archive file, a virtual machine image, or a container image. It is important to note that different stages of the delivery pipeline might mutate an artifact. For example, in the build stage, an input artifact might come from the dependency management stage. In the package stage, the output artifact might be an immutable packaged version of all the things needed to deploy and run the software in an environment.

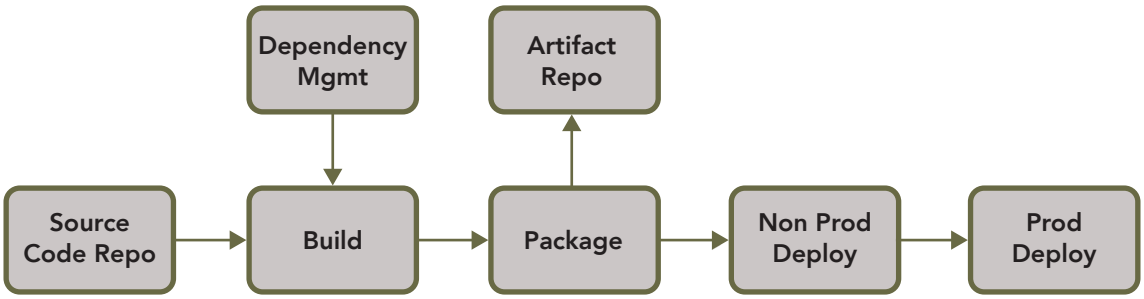


Figure 1: Delivery Pipeline

Creating Better Pipelines

In May of 2018, Capital One wrote the blog post "Focusing on the DevOps Pipeline" explaining what it means to "deliver high quality working software faster."[4] They describe their policy as such:[5]

- *High quality* meaning no security flaws, in compliance, minimum defects, etc.
- *Working* meaning end to end it really works for all parties, that it's been tested, and all dependencies are satisfied.
- *Faster* meaning as soon as possible without sacrificing quality.

The blog post also describes the concepts of "gates," or guiding design principles, later described as "control points:"[6]

At Capital One, they designed pipelines using what they call the 16 Gates. These guiding design principles are as follows:

- Source code version control
- Optimum branching strategy
- Static analysis
- >80% code coverage
- Vulnerability scan
- Open source scan
- Artifact version control
- Auto provisioning
- Immutable servers
- Integration testing
- Performance testing
- Build deploy testing automated for every commit
- Automated rollback
- Automated change order
- Zero downtime release
- Feature toggle[7]

The design ensures that every time software is pushed through the pipeline these control points will be evidenced.

Control points are a form of both metadata and evidence for actions taken during the development, production, and promotion processes. These control points should be defined at every phase of continuous integration and preserved in logs from the build or logs from how an artifact was built. Ultimately, this kind of automated pipeline metadata in the form of control points allows organizations to move to a decentralized form of decision-making, thus moving away from centralized forms commonly used in most enterprises.

DevOps Automated Governance

DevOps practices increase the tempo of software delivery. This creates tension with governance programs that rely on the manual review of artifacts, documents, and

scans. If we can push a change to production every few minutes, no manual process can keep up. So, just as we have automated testing and deployment processes, we must also seek to automate the governance processes. (And just as automated testing and deployment processes reduce variation and manual error, we should also expect automated governance to enjoy the same benefits.)

Many tools have emerged to address pieces of the governance process. We can regard these individual actions as individual controls. To be integrated into an automated governance process, the output from a control must be recorded as an attestation to confirm the control was applied and describe what the control did. Some examples of attestations collected from controls would include statements such as:

- Control: Unit tests
  Attestation: "All tests executed and passed."
- Control: Clean dependencies
  Attestation: "All dependencies in this build satisfy local licensing policies."
- Control: Clean dependencies
  Attestation: "All dependencies in this build are free of known security defects."

A single control may record more than one attestation. These attestations begin as ordinary tool outputs but need to be collected in a way that auditors can later verify the origin and integrity of the attestation. (An example of a mechanism to collect such attestations is the open-source tool Grafeas, discussed in the next section.)

Reference Architecture

Our reference architecture maps a delivery pipeline to specific controls that will produce evidence for collection. This reference architecture offers a starting point for a DevOps automated governance process. Implementers will add to this architecture and adapt it to their particular toolset and delivery pipeline.

Evidence and Automated Traceability

In 2017, Google introduced an open-source initiative called Grafeas (the Greek word for "scribe") to help organizations define a uniform way to audit and govern a modern

delivery pipeline. Grafeas supplies a simple reference architecture for a set of APIs to gather metadata (i.e., control points) in a common model. Another open-source initiative designed for a similar purpose is Capital One's Hygieia.

Consider an example of a control that Grafeas or Hygieia could accept, such as "clean dependencies." It may be applied by a tool such as Sonatype Nexus. Just applying the tool to keep the dependencies clean, however, is not enough to qualify as automated governance. The missing part is evidence that the control was applied at a point in time to a set of inputs. Figure 2 shows how an attestation may be constructed from a set of inputs that connect the output of a control together with the inputs (including implicit inputs, like configuration).

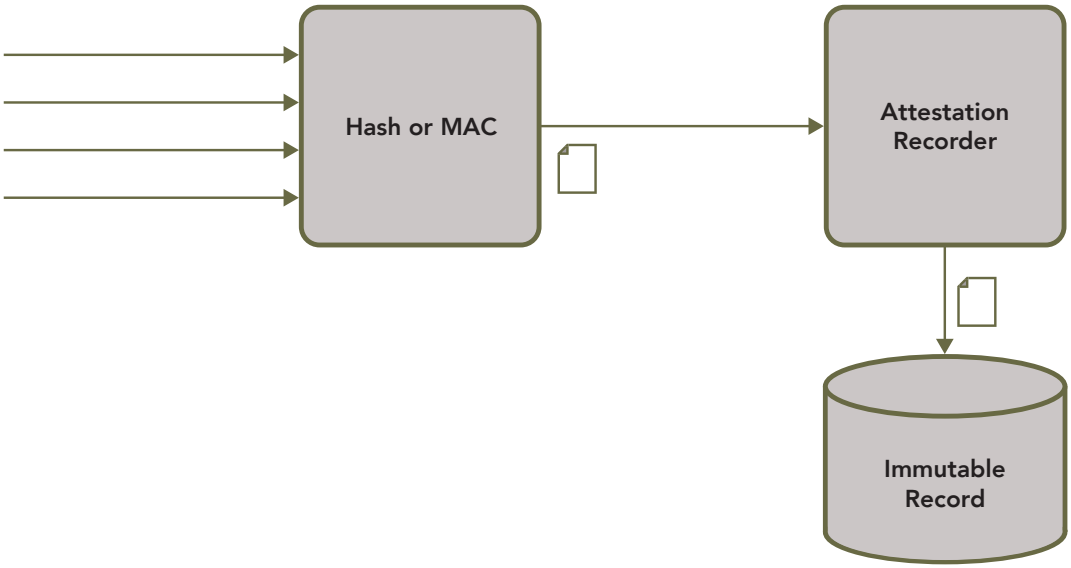


Figure 2: Constructing an Attestation

Hashing and message authentication codes provide tamper resistance to the attestation. This attestation can be recorded immutably. A series of attestations can be connected to reconstruct the flow of a change through the delivery pipeline in much the same way a call tree can be reconstructed from individual spans.

The Model

The model first describes a typical software delivery pipeline. For each stage the model identifies a set of inputs, outputs, actors, and the actions that can occur at that stage.

Next, the model identifies a set of risks that can be attributed to the stage. Finally, based on the identified risks, a set of controls are chosen to mitigate the risks and attest to the input, output, actors, and actions involved. Figure 3 shows the general process of the governance model.

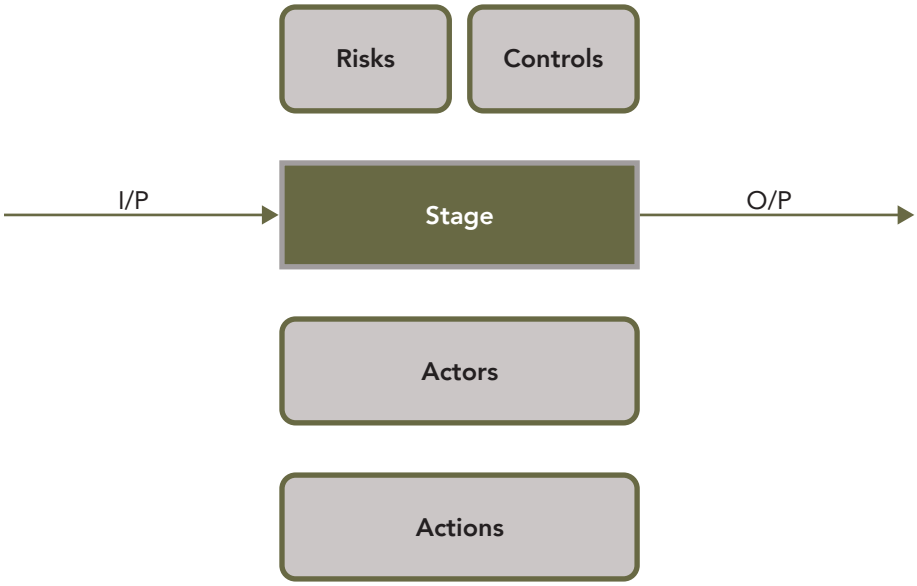


Figure 3: Basic Governance Model

To avoid repetition in every stage, the model factors out a set of common actors and controls that appear in all stages.

Common controls:

- access control
- audit trail/log
- source control
- usage policies

Common Actors:

- auditor; risk/compliance office
- the system itself
- tool administrators

Delivery Pipeline

Currently, software is delivered via an automated and controlled process called a "pipeline." (Figure 1, which you can find earlier in this paper, depicts a general reference for a pipeline.) A typical pipeline is a set of pre-composed stages that integrate with many tools and platforms to automatically send tested changes to production. The pipeline is the heart of an end-to-end delivery life cycle.

Typical pipeline stages and related artifacts are listed below:

- **Source code repository:** A version control tool that hosts all assets related to an application's software and services. Organizations which are mature in DevOps practices use version control for application, infrastructure (as code), tests, and all configurations. (Note that these may not all reside in a single source tree. Application code and production configuration are typically separated according to access rights.) Every change in any code is version controlled. Typically, Git is used to manage this repository.
- **Build:** In this stage, source code is compiled (when a compiled language is used), unit-tested, scanned, and linted for quality and security.
- **Dependency management:** This stage is where external libraries and/or base images (e.g., virtual machines or containers) are stored and from which they are consumed internally. This is the entry point for outside code.
- **Package:** In this stage, the deployable artifact is composed from source code and external dependencies. The artifact may be an archive file, a virtual machine image, or a container image. The resulting package is uploaded to a binary artifact repository.
- **Artifact repository:** This is a version control tool that hosts all packaged artifacts produced via the build and packaging stages. Artifacts in this repository should be immutable.
- **Non-prod deploy:** In this stage the artifact is deployed to one or more non-production environments where various tests are applied. There can be one or more of these stages depending upon the testing needs.
- **Prod deploy:** This is the final stage of a typical pipeline where the tested and approved artifact is finally deployed to the production environment. The actions

in this stage can use a variety of rollout and exposure strategies to make the artifact available for use.

Stage 1: Source Code Repository

Figure 4 shows a generalized overview of what an automated governance model might look like during the source code repository stage.

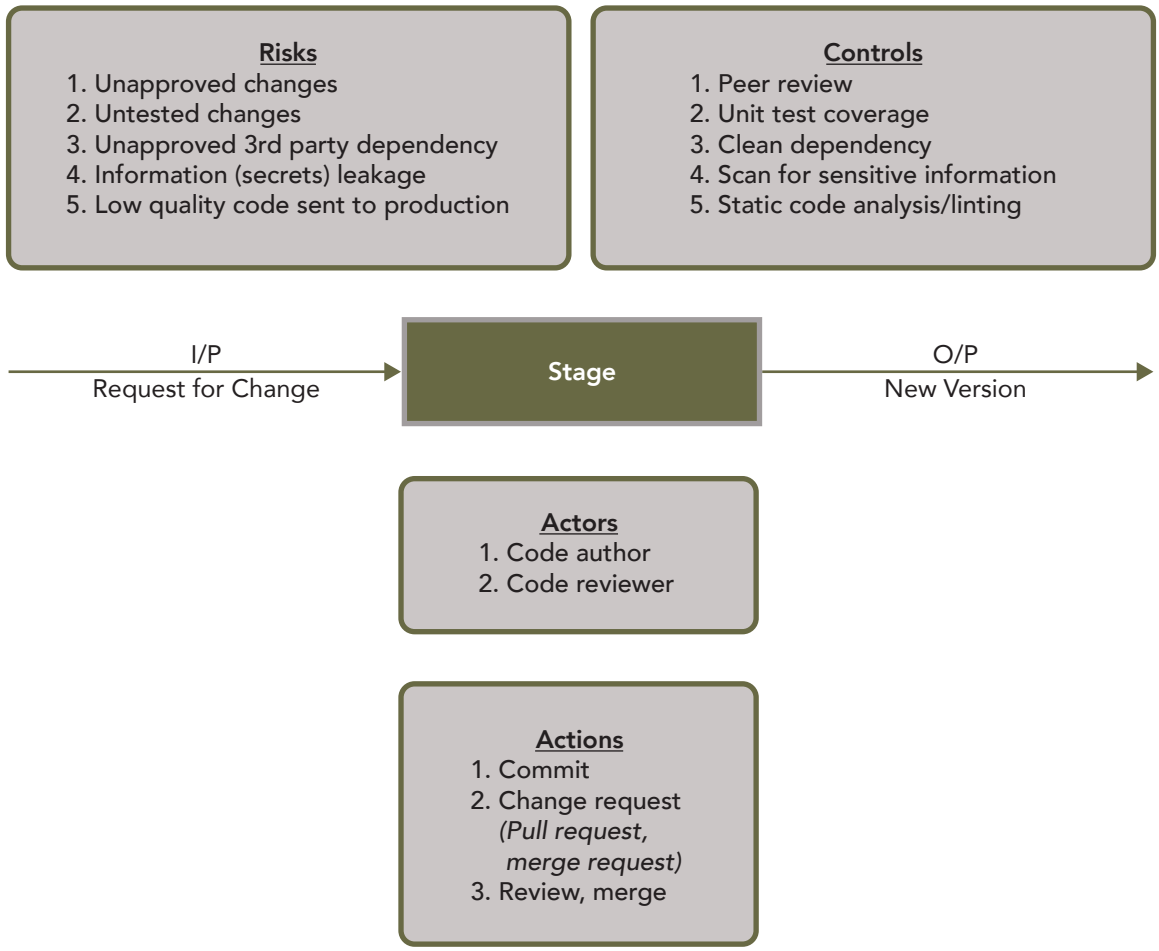


Figure 4: Governance during the Source Code Repository Stage

The primary actors behind automated tools are the code author and the reviewer(s). The actions that instigate the checks associated with this pipeline stage are:

- **Commits:** Triggering pre-commit hooks, unit tests, and dependency management.
- **Pull requests/merge requests:** Triggering static code analysis, unit tests, and code review.

Input

- **Request for change:** A change in source code is initiated by a request for change; a request for change can be a new feature request, a bug fix request, or a request for refactoring or redesign.

Output

- **New version:** A new version of the code base. In Git terms, it is the new SHA.

Actors

- **Code author:** The person who is making the actual code change.
- **Code reviewer:** The person reviewing code changes.
- **Repository admin:** Also known as the owner(s) of the code repository, this person is also responsible for merging the code change to the main code branch.

Actions

- **Code commit:** The actual code change pushed to a temporary place (such as a fork or a branch).
- **Code review:** Peer reviewing code changes.
- **Change request:** A request to merge the changed code to the main branch. In GitHub, this is called a pull request.

Risks

- **Unapproved changes:** Unapproved and unreviewed changes may cause degraded and/or unwanted behavior of the service.
- **Untested changes:** Code changes that are not tested may cause degraded and/or unwanted behavior of the service.
- **Unapproved third-party dependency:** Unapproved dependencies can introduce legal and security vulnerabilities in the software.

- **Sensitive information leakage:** Accidentally including sensitive information (e.g., credentials, non-public customer information, system account details, etc.) in the source code can result in legal risk, security incidents, or data breach.
- **Low quality code sent to production:** Even though new code functions as expected, low quality code may cause operational issues, technical debts in terms of code manageability, extensibility, complexity, etc.

Controls

We consider a source control scheme that involves a single master branch and the use of feature branches to isolate and control code contributions. In order to merge the changes captured by a branch into master, it is typical to require that the code passes a number of checks.

- **Peer-based code review** (e.g., via GitHub Pull Requests) has been shown to have the greatest impact on code quality.
- **Unit test coverage** (e.g., via SonarQube) is typically tracked, as untested functionality tees up significant risk when refactoring, optimizing, or altering API usage.
- **Clean dependencies** (e.g., via Sonatype Nexus) refers to a check that open-source dependencies satisfy enterprise-level licensing guidelines and are free of known vulnerabilities.
- **Information leakage analysis** (e.g., via GitHub pre-commit hooks to grep for sensitive tokens) checks that passwords, access tokens, and other types of sensitive information are not being checked into a repository.
- **Static code analysis** (e.g., via MuseDev) involves statically scanning for performance, reliability, and security issues as part of the merge decision for new code.

Stage: Build

Figure 5 shows a generalized overview of what an automated governance model might look like during the build stage.

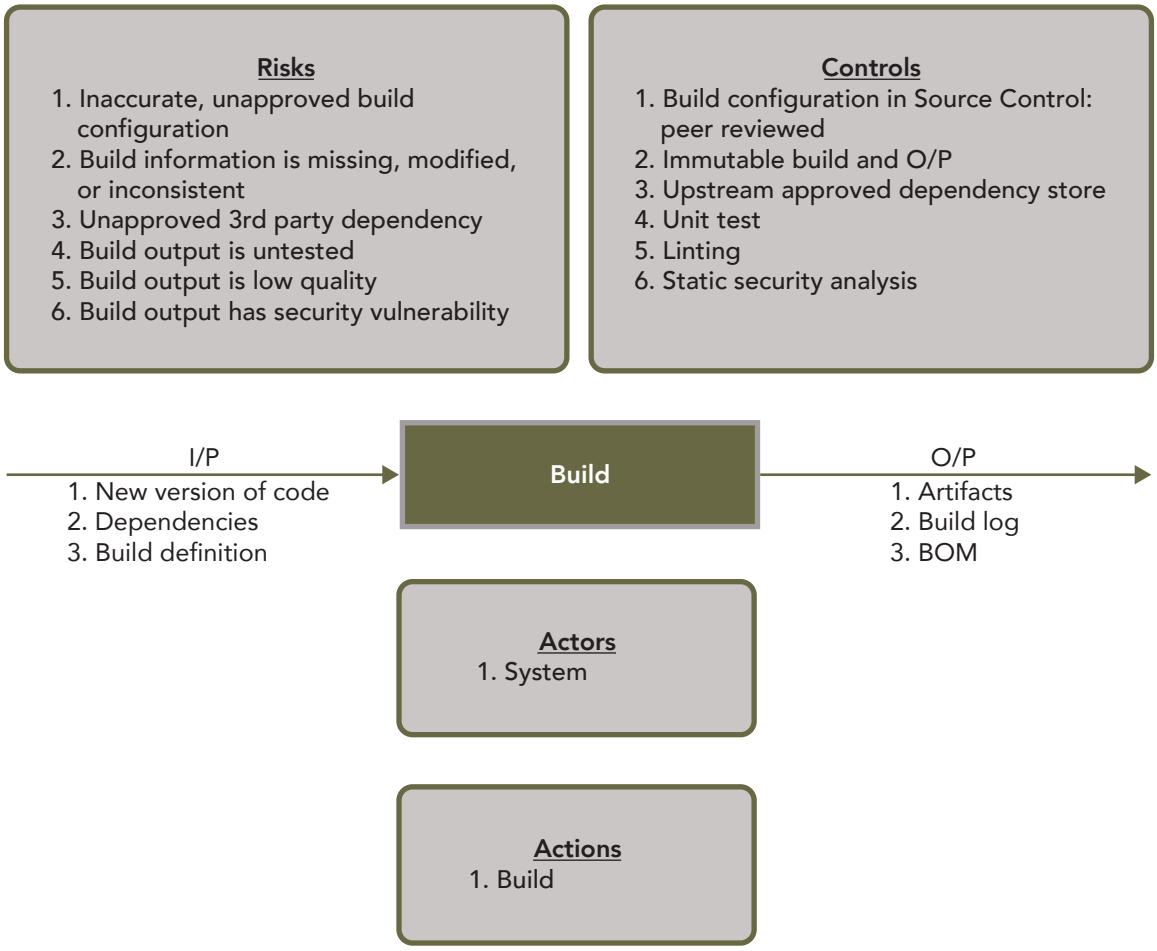


Figure 5: Governance during the Build Stage

Input

- **New version of code:** Software is rebuilt on demand or automatically when the source code version is changed.
- **Dependencies:** In many situations, when dependency versions change, software needs to be rebuilt.
- **Build definition:** Build definition, also known as build script, contains the codified build steps.

Output

- **Artifacts:** The main components of the deployment package that are sent to production

Actors

- **System:** Usually the build stage is triggered by a change in the source code. There is no manual intervention needed in this case.

Actions

- **Build:** The only action in this stage is execution of the build definition.

Risks

- **Inaccurate, unapproved build configuration:** An inaccurate, unapproved build configuration may produce incorrect build artifacts.
- **Missing, modified, inconsistent build information:** The build artifact might not be traceable, authentic, or reproducible.
- **Unapproved third-party dependency:** Inclusion of unapproved third-party dependencies in the build stage may result in legal and security vulnerabilities.
- **Build output is untested:** The build output, when deployed to production, may not function as expected.
- **Build output has security vulnerability:** The build output may contain a security vulnerability.

Controls

- **Build configuration in source control and peer reviewed:** As a basic DevOps practice of having "everything as code," build configurations should be source controlled and peer reviewed just as the application code.
- **Immutable build and build output:** To ensure that a build cannot be modified after the fact, every build and the output of the build should be immutable. If any build fails for some reason or the build output is unreliable, a fresh build should be initiated.
- **Upstream approved dependency management system:** To ensure that every dependency downloaded is approved for use, the build system should be restricted to use only on an approved dependency management system.
- **Unit test:** Every build should include unit test execution and should complete successfully only if the unit test pass rate and coverage meet predefined criteria.
- **Linting:** Every build should scan the source code for code quality and should complete successfully only if the analysis result meets predefined criteria.

- **Static security analysis:** In addition to unit test execution and linting, source code also should be scanned for potential security vulnerabilities. Due to the limitations of available tools, it may not be possible to execute a static security scan for every build. However, an out-of-the-band scan on a periodic basis should be in place. Many organizations execute a full static security scan daily, or at least before production release.

Stage: Dependency Management

Figure 6 shows a generalized overview of what an automated governance model might look like during the dependency management stage.

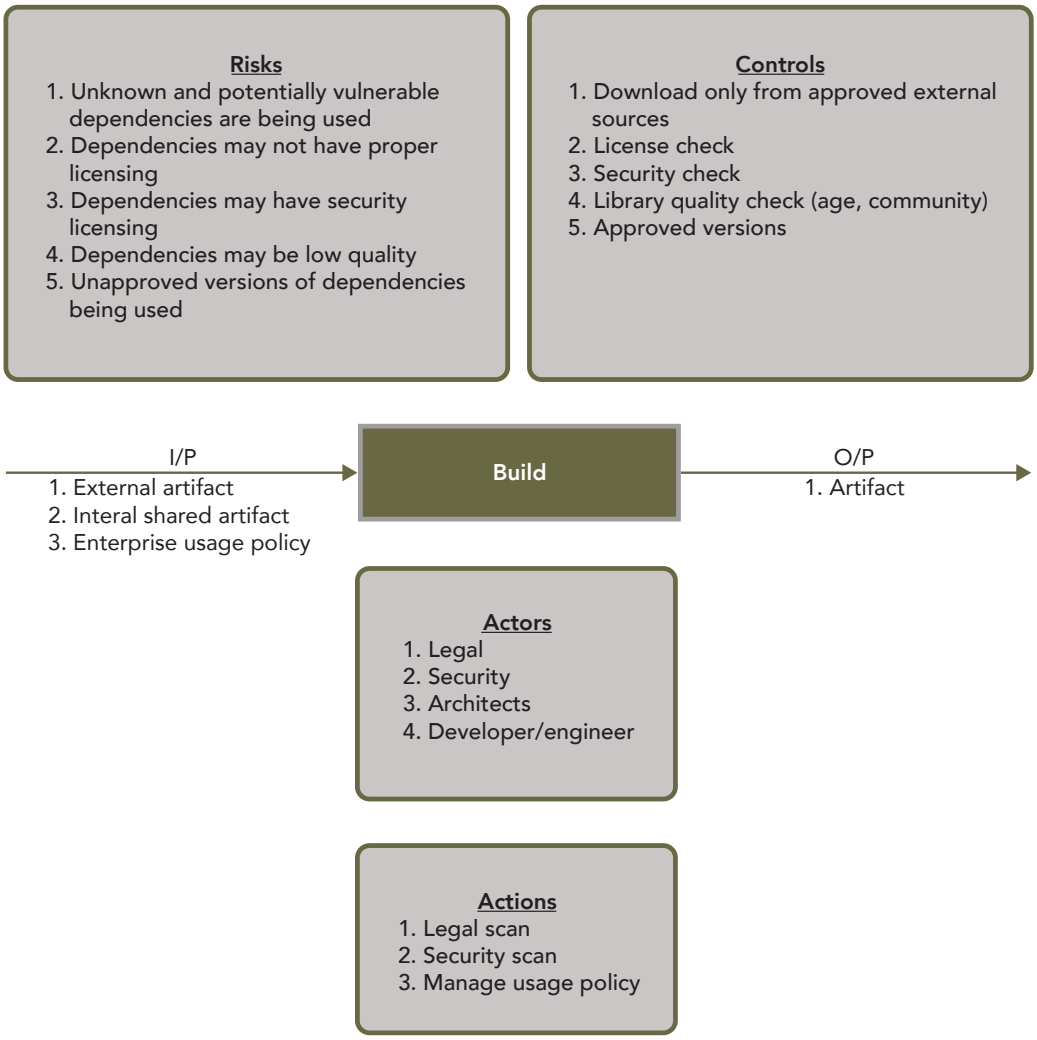


Figure 6: Governance during the Dependency Management Stage

Input

- **External artifacts:** These are dependencies that have been downloaded from external repositories.
- **Internal shared artifacts:** Many dependencies are produced internally and shared among others internally.
- **Enterprise usage policies:** Many enterprises maintain specific usage policies for specific types of applications. For example, the enterprise might have a policy that software distributed to customers may not use dependencies with "copy-left" licenses.

Output

- **Artifact:** Artifact that was requested by the build system.

Actors

- **Legal:** Enterprise legal teams create policies based on legal requirements.
- **Information security:** Security teams create policies based on security requirements.
- **Architects:** Architects create policies based on architectural requirements that include the health of the dependencies (e.g., age, popularity, activity status, etc.).
- **Developers/engineers:** Developers and engineers are the consumers and creators of dependencies.
- **System:** Systems, such as build systems, download dependencies.

Actions

- **Legal scan:** Dependencies are scanned for legal vulnerabilities.
- **Security scan:** Dependencies are scanned for security vulnerabilities.
- **Manage usage policies:** Legal, security, and architecture teams create dependency usage policies.

Risks

- **Unknown and potentially vulnerable dependencies are in use:** One of the biggest risks in software development is the risk of unknowns. This includes the risk of using unknown dependencies that can cause damage in many forms.

- **Dependencies may not have proper licensing:** Using incorrectly licensed dependencies can lead to legal issues.
- **Dependencies may have security vulnerabilities:** Using dependencies with security vulnerabilities is a huge risk across every enterprise.
- **Dependencies may have low quality:** Using low-quality dependencies leads to low-quality software.
- **Unapproved versions are in use:** Using unapproved versions of dependencies may result in security or legal issues. This may also cause systems in production to behave unexpectedly.

Controls

- **Download only from approved external sources:** Every enterprise should create a list of trusted sources of their dependency needs.
- **License check:** Dependencies that are downloadable from the dependency management system should have licenses that satisfy enterprise legal requirements and usage policies.
- **Security check:** Dependencies that are downloadable from the dependency management system should meet enterprise security requirements and satisfy usage policies.
- **Dependency quality check:** Dependencies that are downloadable from the dependency management system should meet architecture standards and should satisfy usage policies.
- **Approved versions:** Only approved versions of dependencies are made available.

Stage: Package

Figure 7 shows a generalized overview of what an automated governance model might look like during the package stage.

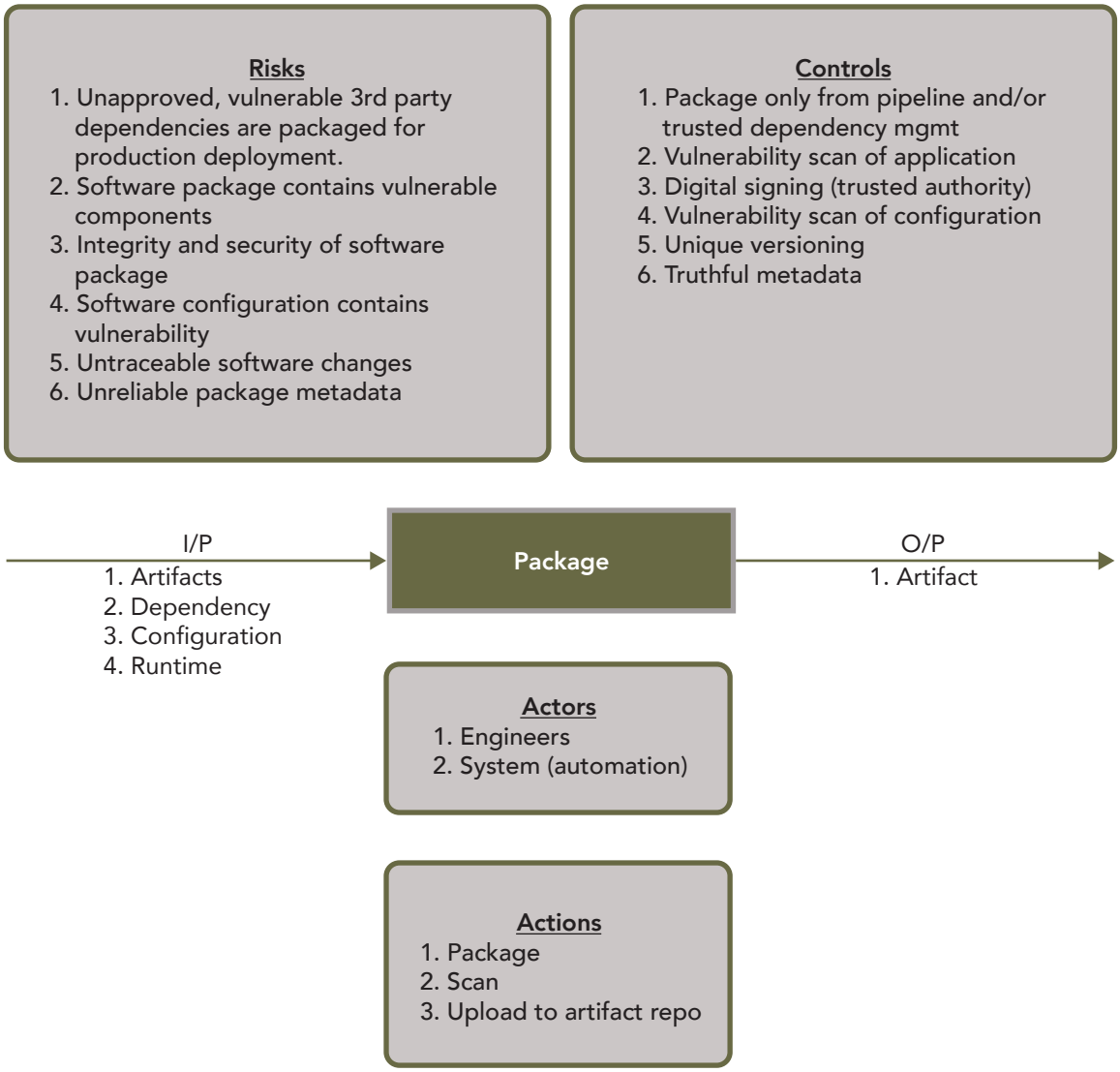


Figure 7: Governance during the Package Stage

Input

- **Artifacts:** The build artifacts that need to be packaged.
- **Dependency:** Dependencies that are not packaged in build artifacts
- **Configuration:** Configurations that are required to run the software in an environment
- **Runtime:** Any runtime that should be packaged for deployment. This may include base images, virtual machines, etc.

Output

- **Artifact:** At this stage, the artifact is a packaged version of all the elements needed to deploy and run the software in an environment.

Actors

- **Engineers:** Developers, operations admin, and system admin who contribute to the packaging process.
- **System:** The automated way in which the packaging step is executed

Actions

- **Package:** The automated process that creates a deployable artifact.
- **Scan:** The process to scan the deployable artifact to detect legal and security vulnerabilities.

Risks

- **Unapproved, potentially vulnerable third-party dependencies are packaged in the deployable artifact:** Third-party dependencies downloaded during the packaging process may contain vulnerabilities that cause legal and security issues.
- **Components with vulnerabilities are packaged in the deployable artifact:** Internal components produced by the build's vulnerabilities may cause security issues.
- **Software configuration contains vulnerabilities:** Even though the actual software may not have vulnerabilities, configurations can contain data that do not meet security standards. These may cause security issues.
- **Untraceable software changes:** Packaged artifacts containing changes that cannot be traced back to source code or approved dependencies may cause unpredictable behavior in the software.
- **Unreliable metadata:** Unreliable or missing artifact metadata may cause confusion and at times can cause incorrect software to be deployed in production.

Controls

- **Packaging only from trusted dependency sources:** Packaging system should download dependencies only from trusted dependency sources.

- **Vulnerability scanning:** Even trusted dependency management sources may contain dependencies with vulnerabilities. New vulnerabilities are discovered every day. The packaging system should execute a vulnerability scan just like the build system against the latest vulnerability data to detect vulnerabilities that were potentially undetected during the build stage.
- **Digital signing:** The packaging system should digitally sign the deployable artifact to ensure authenticity.
- **Artifact versioning and metadata:** Every artifact produced by the packaging system should be immutable and versioned with an approved versioning scheme. The packaging system also should add metadata to the deployable artifact.

Stage: Artifact Repository

Figure 8 shows a generalized overview of what an automated governance model might look like during the artifact repository stage.

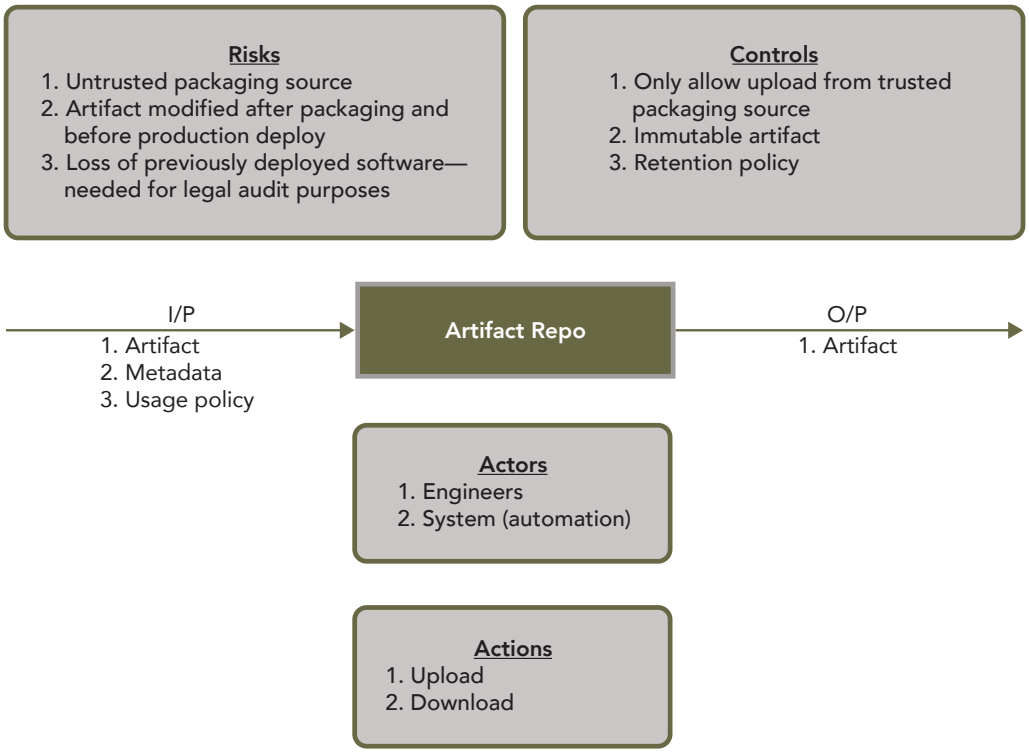


Figure 8: Governance during the Artifact Repository Stage

Input

- **Artifact:** The artifact uploaded by the packaging system and downloaded during the deployment stage.
- **Metadata:** The metadata to add to the artifact.
- **Usage policy:** Enterprise policy on the upload and download of artifacts.

Output

- **Artifact:** The artifact downloaded by the deployment system.

Actors

- **Engineers:** Developers, operations admin, and system admin who contribute to the packaging process.
- **System:** The automated way in which the packaging step is executed.

Actions

- **Upload:** Uploading of artifacts by packaging system.
- **Download:** Downloading of artifacts by deployment system.

Risks

- **Untrusted packaging store:** An unknown or untrusted packaging store can upload vulnerable and/or unapproved artifacts.
- **Artifact modified after packaging and before deployment:** If an artifact can be modified before deployment, there will be no assurance of the integrity of what will be deployed.
- **Loss of previously deployed artifact:** Most enterprises need to archive older versions of software to meet legal and regulatory requirements

Controls

- **Only allow upload from trusted packaging source:** Configure the artifact repository to accept upload requests only from known and trusted packaging sources. Many enterprises restrict individual users from uploading to the artifact repository.
- **Immutable artifact:** No artifact in the repository can be overwritten; only a newer version of the same artifact can be uploaded.

- **Retention policy:** The artifact repository should implement a retention policy for all released artifacts.

Stage: Non-Prod Deploy

Figure 9 shows a generalized overview of what an automated governance model might look like during the non-prod deploy stage.

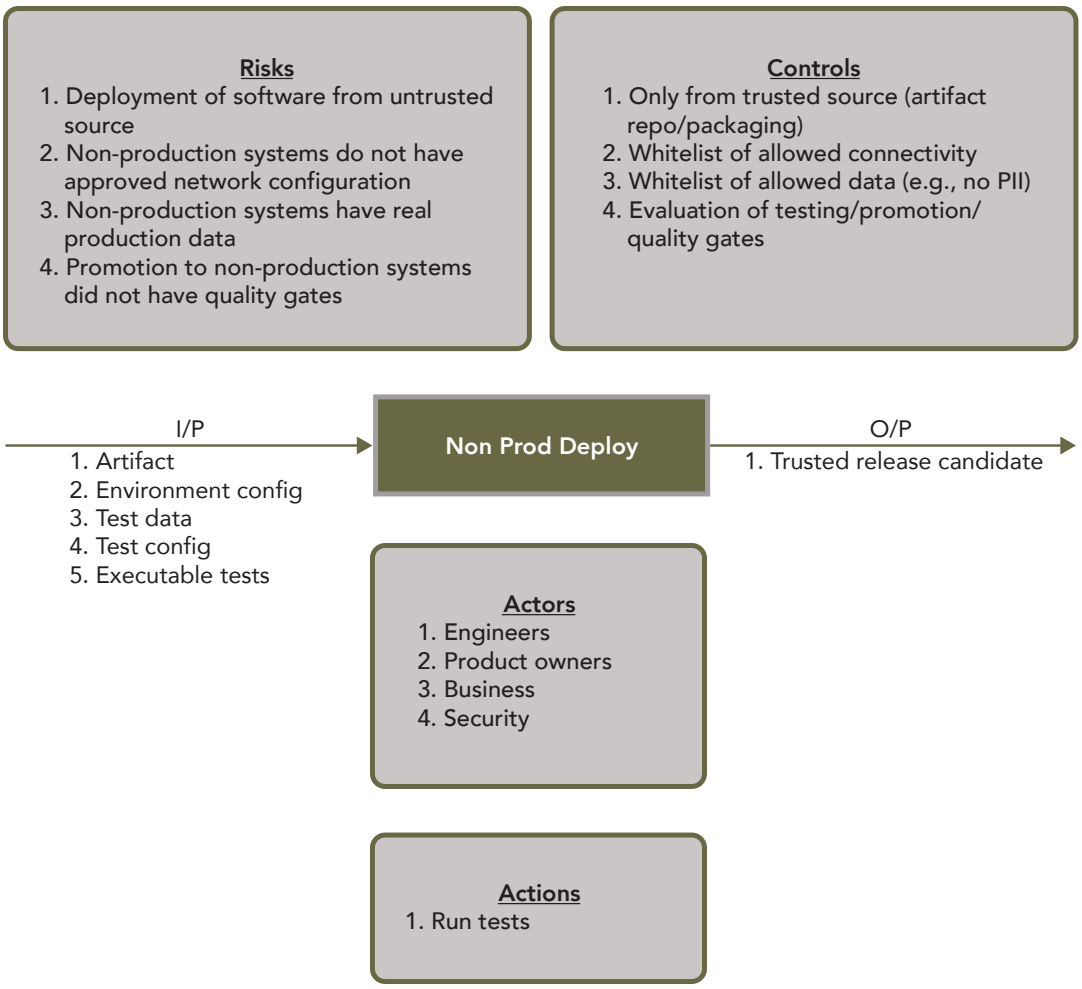


Figure 9: Governance during the Non-Prod Deploy Stage

Input

- **Artifact:** The artifact that will be deployed.
- **Environment configuration:** Any environment configuration that was not or cannot be packaged.

- **Test data:** Any test data set needed to execute tests in the non-production environment.
- **Test configuration:** Any configuration needed to execute test cases in the non-production environment.
- **Executable tests:** Tests that can be executed in the non-production environment

Output

- **Trusted release candidate:** A successful completion of this stage produces a release candidate that can be trusted, provided all controls in previous stages were in place and followed.

Actors

- **Engineers:** The developers, operations admin, and system admin who contribute to the packaging process.
- **System:** The system executes the packaging step in an automated way.
- **Product owners, business partners:** Product owners involved in testing functionalities in the non-production environment.
- **Information security:** Information security team may run security related tests in non-production environment

Actions

- **Deployment:** The process that deploys an artifact.
- **Run tests:** Various types of tests executed in the non-production environment

Risks

- **Deployment of artifact from untrusted sources:** There is the risk of testing the wrong software before producing a release candidate.
- **Non-production systems with unapproved network configuration:** With an unapproved network configuration, there is the risk of executing tests with unpredictable results or untrusted results.
- **Non-production systems with production data:** In many enterprises, non-production systems should never have production data due to legal and privacy risks.

- **Promotion to non-production systems did not have quality gates:** Without a set of checks, or quality gates, there is the risk of producing a release candidate that has low quality, and can potentially produce unpredictable results in the production environment.

Controls

- **Fetch artifact only from trusted source:** The non-production deployment stage should fetch deployable artifacts only from trusted sources (such as the enterprise's artifact repository).
- **Whitelist of allowed connectivity:** The connections allowed in the non-production deployment stage should be reviewed and kept up-to-date. Pre-approved connections on the whitelist should not be allowed.
- **Whitelist of allowed data:** The non-production deployment stage should have access to only a set of whitelisted test data. This data should not contain any real customer data or sensitive information.
- **Quality gate evaluation:** Non-production deployment should be executed only if it meets a set of predefined criteria (e.g., 100% test pass rate with 80% coverage, no new high severity security vulnerability, etc.). The quality gate also should consider the drift and difference between production and non-production environments; the non-production environment should mimic the production environment.

Stage: Prod Deploy

Figure 10 shows a generalized overview of what an automated governance model might look like during the prod deploy stage.

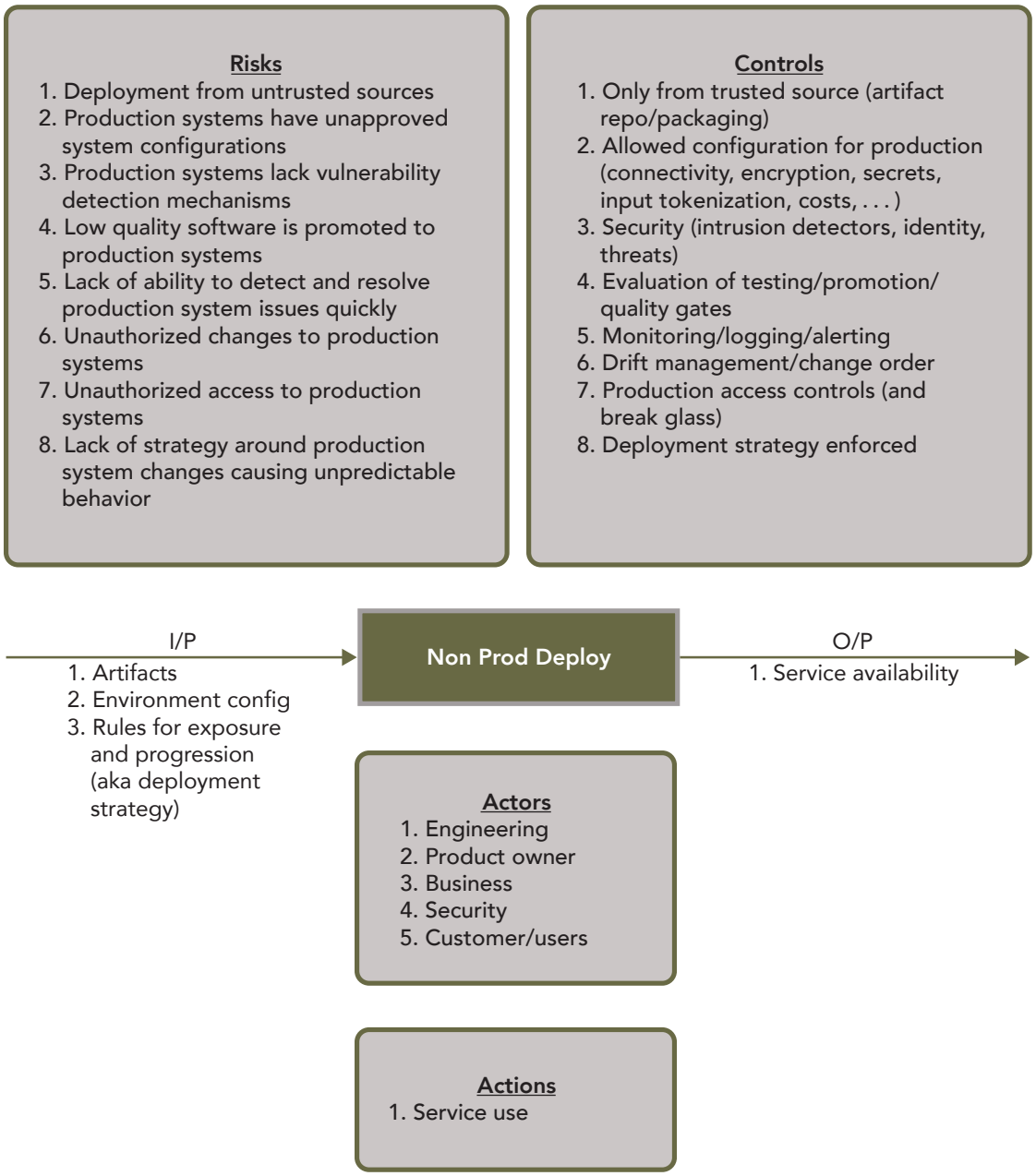


Figure 10: Governance during the Prod Deploy Stage

Input

- **Artifact:** The artifact that will be deployed.
- **Environment configuration:** Any environment configuration that was not or cannot be packaged.

- **Deployment strategy:** A deployment strategy that is scripted and/or documented.

Output

- **Service availability:** Availability of service with expected behavior.

Actors

- **Engineers:** The developers, operations admin, and system admin who contribute to the deployment process.
- **System:** The system executes the deployment stage in an automated way.
- **Product owners, business partners:** The product owners involved in making decisions of production release readiness and testing out functionalities in the production environment.
- **Information security:** The information security team runs security related checks in production environment.
- **Customers/users:** Who uses the service.

Actions

- **Production deployment:** Execution of the deployment process.

Risks

- Deployment from untrusted sources.
- Production systems have unapproved configuration.
- Production systems lack vulnerability detection mechanism.
- Low quality software deployed to production.
- Lack of ability to detect and resolve production issues.
- Unauthorized changes to production systems.
- Unauthorized access to production systems.
- Lack of strategy around production system changes causing unexpected behavior.

Controls

- **Fetch artifact only from trusted source:** The production deployment stage should fetch deployable artifacts from only trusted sources (such as the artifact repository).

- **Only use approved configurations:** The production system should use an approved set of configurations, such as network connectivity, encryption, tokenization, secrets management, etc.
- **Security monitoring:** The production system should have intrusion detection, threat monitors, and other approved security mechanisms. It should also have approved monitoring, logging, and alerting mechanisms.
- **Change management:** The production system should have an automated change management mechanism.
- **Access control:** The production system should have a strict access control mechanism. By default, no one should have access to production systems except by means of a break-glass mechanism.
- **Deployment strategy enforced:** Production deployment should enforce deployment strategy (e.g., deploy only during a specified time window, use blue-green deployment, use canary deployments, etc.).

Recording Attestations

Example with Universal Metadata API

In practice, each automated governance solution will be contextual to each organization and will require different considerations and controls. For example, organizations that require compliance with self-identifying risk control self-assessments might need extensive design for functional and non-functional test acceptance. When digesting some of the intense complications driven from compliance, audit, and risk, each organization must do their due diligence in creating a solution that is personalized to their needs.

In this example, we assume the software delivery pipeline uses the following practices:

- development for a microservice application with a Java component
- trunk-based development
- container-based application with Kubernetes for container orchestration and deployment
- continuous release of application deployment with a canary release strategy

Here, we provide an example framework on how to capture universal metadata. This example uses an API framework to capture and store the attestations. The attestation model utilized consists of two parts: notes and occurrences. A note is an abstract view of a piece of metadata. Each note will represent a control (such as a pull request, peer review, etc.).

Note Example:

```
{
    "name": "example_project/peer_review/note",
    "shortDescription": "Approved commit record with documented
        approver.",
    "attestationAuthority": {
        "hint": {
            "humanReadableName": "github"
        }
    }
}
```

An occurrence is an instantiation of a note that describes the details for a given note. For the static security scan as a note, for example, vulnerabilities could be identified as occurrences of evidence.

Occurrence Example:

```
{
        "resourceUrl": "${RESOURCE_URL}",
    "noteName": " example_project/peer_review/note",
    "attestation": {
        "peer_review": {
        "repo": {"id": XXXXXXXXX},
        "pull_request": {
            "url": "https://api.github.com/repos/.../pulls/1",
```

```
            "id": XXXXXXXXX
            }
        "approved_by": XXXXXXXX",
        "merge_commit": {"hash":"XXXXXXX" }
        }
    }
}
```

The automated governance reference architecture consists of common pipeline components. This example uses Grafeas as the metadata API framework and can be used by highly governed organizations that might require a controlled and secure way of validating attestations, such as providing access control to roles depending on metadata producers and consumers. Figure 11 shows different types of automatic governance.

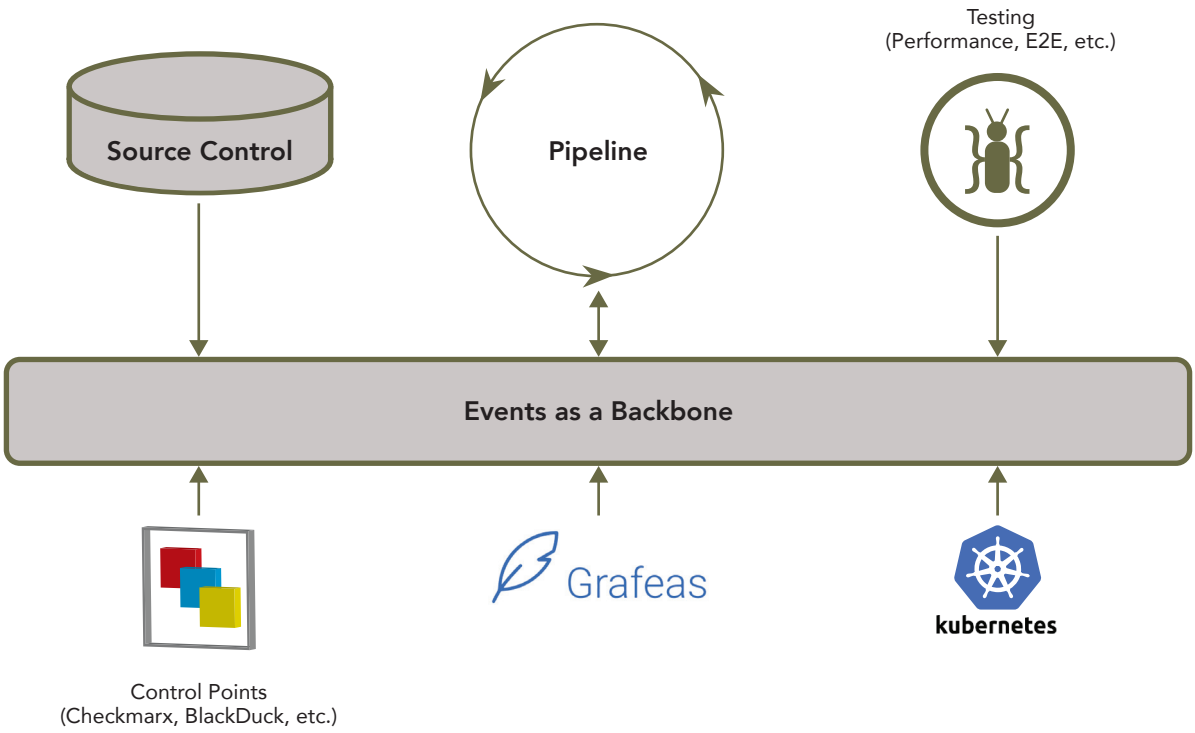


Figure 11: Different Types of Automatic Governance

The architecture will also include an event-driven streaming architecture. In an event stream, a key is used to create logical groupings of events as a stream. The events that drive the event stream are the data within the pipeline defined by the reference architecture. Identifying what key to use will depend on the use case and dependencies. In our example, we utilize build identifiers as well as different event types (e.g., pull request identifiers, vulnerability findings, test results, etc.). A stream processor is used to construct the flow from other streams to build an attestation model. This architecture also offers the ability to guarantee delivery and not lose events during other system outages. This process is shown in Figure 12.

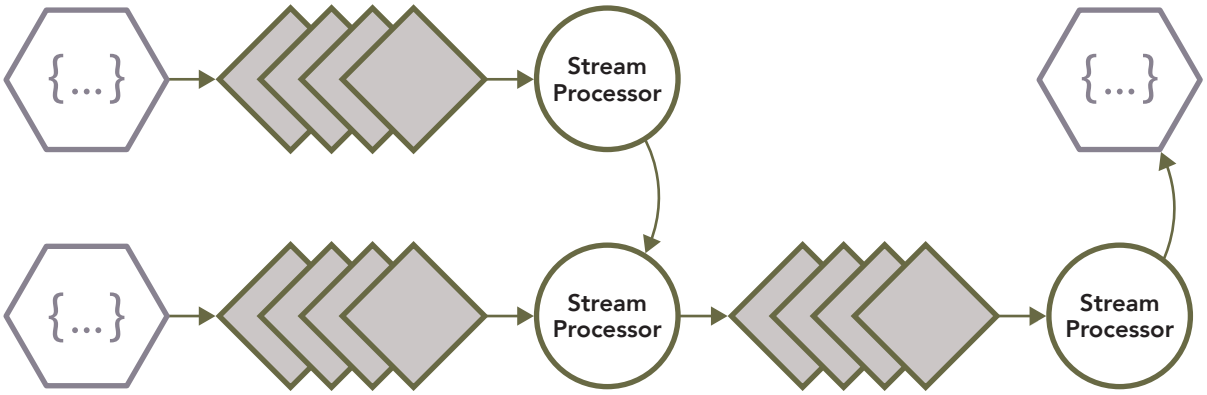


Figure 12: Creating an Event Stream

Now that we have an ability to capture events, store attestations, and read/enforce/report on attestations for a given release, we need to identify the control sources. This example provides a list of control sources that will vary between company, solutions, and providers. Your enterprise might use different control sources from the ones identified in Table 1.

Stage	Control	Example Control Source	Integration	Elements
Source Code Repo	Pull Request	GitHub	Webhook	pull_request repository
Source Code Repo	Peer Review	GitHub	Webhook	actor pull_request repository
Source Code Repo	Unit Test	SonarQube	Pipeline	new_coverage

Source Code Repo	Clean Dependency	Artifactory	Pipeline	dependency source
Source Code Repo	Information Leakage	GitHub	Webhook	(custom)
Source Code Repo	Static Code Analysis	Muse	Webhook	pull_request repository
Build	Build Definition	Jenkins & GitHub	Pipeline	TBD
Build	Immutable Build	Jenkins	Pipeline	TBD
Build	Upstream Approved Dependency	Artifactory	Jenkins	TBD
Build	Unit Test	SonarQube	Jenkins	TBD
Build	Linting	SonarQube	Jenkins	TBD
Build	Static Security Analysis	Checkmarx	Jenkins	TBD
Package	Trusted Dependency Store	Artifactory	Jenkins	TBD
Package	License Check	Artifactory	Jenkins	TBD
Package	Vulnerability Scan	Aqua	Jenkins	TBD
Package	Trusted Authority	Artifactory	Jenkins	TBD
Package	Versioning	Artifactory	Jenkins	TBD
Package	Usage Policy	Artifactory	Jenkins	TBD
Non-Prod Deploy	Trusted Source	Artifactory	Jenkins	TBD
Non-Prod Deploy	Whitelist Connectivity	Istio	Jenkins	TBD
Non-Prod Deploy	Whitelist Data	DBMaestro	Jenkins	TBD
Non-Prod Deploy	Quality Gates	JMeter, Karate, WebDriver	Jenkins	TBD
Production Deploy	Trusted Sources	Artifactory	Jenkins	TBD
Production Deploy	Trusted Configurations	GitHub	Jenkins	TBD
Production Deploy	Intrusion Detection	TBD	Jenkins	TBD
Production Deploy	Monitoring & Alerting	Elastic, PagerDuty	Jenkins	TBD
Production Deploy	Change Management	ServiceNow	Jenkins	TBD
Production Deploy	Secrets Management	Vault	Jenkins	TBD

Production Deploy	Unauthorized Change Detection	Jenkins	Jenkins	TBD
Production Deploy	Production Access Control	Vault	Jenkins	TBD
Production Deploy	Deployment Strategy	Jenkins, Helm	Jenkins	TBD

Table 1: Selected Control Sources

As this table is a work in progress, the TBDs show areas where more research is still needed.

Grafeas can be used to query and report back governance status. The source for metadata storage should be able to query attestations and allow developers, product owners, and organizational leadership to shift left in identifying development gaps for governance. Most importantly, this data can provide traceability for risk, audit, and compliance partners.

Enforcement will utilize Kubernetes with a webhook that calls the Grafeas API to retrieve the required occurrences of attestations after the production deploy. Although the example suggests an ability for continuous release, the same model can also assert a scheduled change approval through a digital signature in a change control system to include manual releases as well.

Extending Enforcement to Audit

As the number of software deployments continues to grow and pipeline complexity exponentially increases, our design patterns must change to support these new norms. DevOps teams need to support real-time validation of compliance in a method similar to the application monitoring practice during production. The DevOps team should empower auditors via systems and tools instead of manually walking auditors through the pipeline. A set of standards will be agreed upon and externally monitored to verify compliance. External monitoring will ensure the separation of duties is maintained and verified throughout the software life cycle.

An auditing system should have the following four qualities:

1. The system is usable by non-technical and technical auditors alike.
2. Each control, actor, and action should be uniquely verifiable.
3. Software should be traceable through all stages without DevOps intervention.
4. The audit log should be immutable.

To satisfy conditions 1 and 3, we will use a red or green light in a web UI. To satisfy condition 2, we will use private keys to sign each step. To satisfy condition 4, we will use a hashing algorithm that maintains separation of duties by copying data to a database that the auditors control.

Architecture

To store information, we will use three separate servers. The first is a system of record that contains transaction data. This includes controls, actors, actions, and output. The second is a system of record that contains private and public keys, steps, and a final hash. Finally, the third is a monitoring system that displays a red or green light by the build ID. This system is illustrated below in Figure 12.

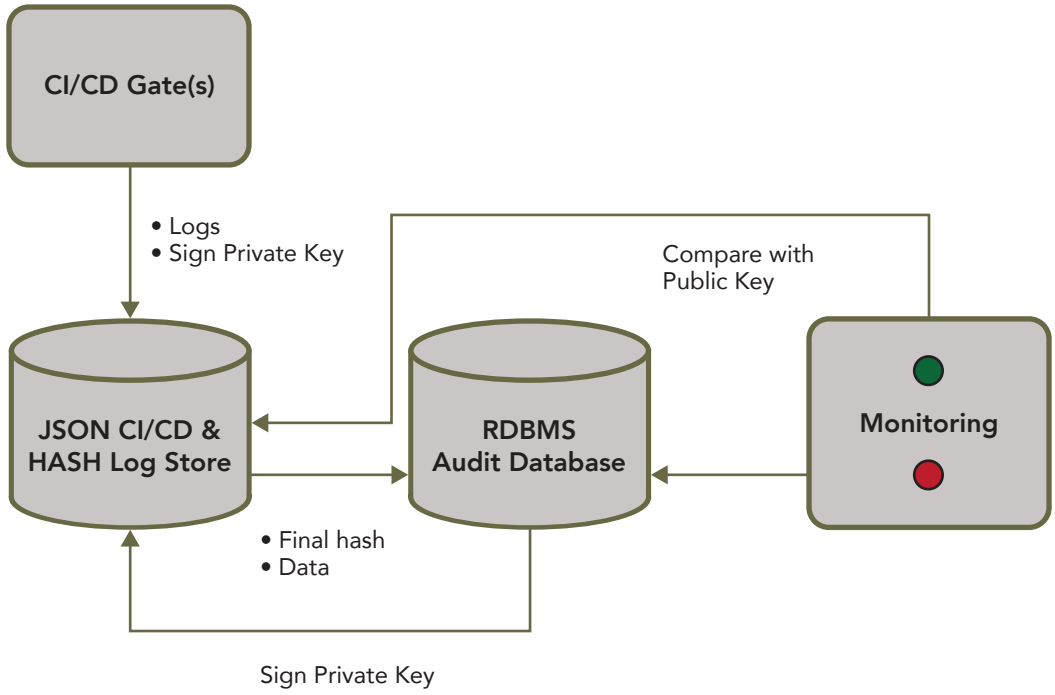


Figure 13: Storing Data in an Automated Governance Reference Architecture

A JSON database will be used to store detailed information about build steps. This database could have multiple roles within the company. For example, it could be used to produce metrics, debug pipeline issues, or produce Slack alerts.

The JSON below is a sample of how you could incorporate the parent hash into your JSON store.

```
{
 “buildID”:”number”,
 “buildDesc”:”string”,
 “transactionID”:”string”,
 “transaction”:[
        {
        “type”:”control, actor, output”,
        “description”:”string”,
        “parentHash”:”string”,
        “currentHash”:”string”,
        “timestamp”:”string”,
        “pass”:”boolean”
        },
        {
        “type”:”control, actor, output”,
        “description”:”string”,
        “parentHash”:”string”,
        “currentHash”:”string”,
        “timestamp”:”string”,
        “pass”:”boolean”
        }
 ]
}
```

Auditors would own the auditing database. This database would be used to verify DevOps pipeline compliance with agreed upon standards. Two sample databases are shown in Figure 13.

Build Hashes	
Hash	Primary Key
BuildID	String
Timestamp	DateTime
Step	String
PassOrFail	Boolean
Final	Boolean

Keys	
ID	Primary Key
Type	String
Timestamp	DateTime
PrivateKey	String
PublicKey	String

Figure 14: Sample Auditing Database Tables

The final piece in this proposed architecture is the monitoring system. The monitoring system uses two colors to signify authenticity of audit logs. A green circle means the keys align and auditors can trust the output of the CI/CD pipeline JSON database. A red circle means auditors cannot trust the output of one or more build steps. The auditor can verify each step, if required. On the other hand, this verification system gives auditors a quick mechanism to scan for compliance. This monitoring system would indicate:

- The overall trustworthiness of all pipelines (green would mean all keys align).
- The overall trustworthiness of a full build (green would mean build keys align).
- The trustworthiness of a stage or step (green would mean a stage or step build keys align).

Hashing and Keys

A hashing algorithm is used in our example architecture to verify and maintain the audit log state. A SHA256 digital key is associated with each control point to maintain security. The hashing algorithm takes multiple inputs and signs them with a digital key to create a random hash of the output, which can be quickly verified via a public key on the monitoring server. The generated hash is inserted as data in the next hashing step. This is repeated for each stage in the software lifecycle. Once complete, the final hash is inserted into the auditing database. This process is illustrated in Figure 14.

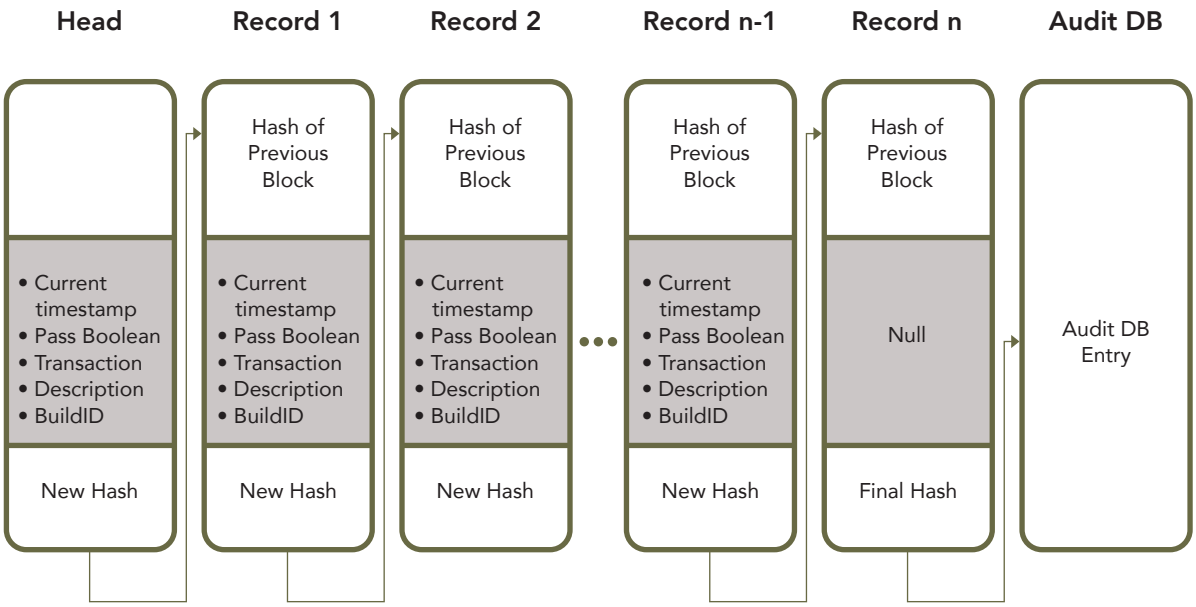


Figure 15: Hashing Algorithm

The final hash can quickly determine if audit logs have been tampered with by working backward and/or verifying the final hash.

A sample of entries, such as those below, could be used to create a hash:

- the current timestamp
- hash of the previous block
- pass Boolean
- transaction
- transaction ID
- description
- build ID
- timestamp

Summary

Creating Trust in the Deployment Pipeline

With the advent of DevOps practices, more and more of the delivery pipeline is being automated and decentralized. However, with these new automated and decentralized models, organizations need to ensure common validation and trust mechanisms throughout the continuous update process. In other words, an optimized process creates a signed output to authenticate the software development. Approved signatures would be part of the automated pipeline process. This would give an organization assurances that the automated continuous updates are certified by a known authority. The process aims to create trust in an organization's delivery pipeline.

Extending Automated Governance throughout the Software Supply Chain

This paper addresses the delivery pipeline. While this is a critical step in the overall software supply chain, there is much room left to extend automated governance. We regard this paper as the minimum viable product of automated governance. We look forward to your feedback and to future work extending these techniques across organizational boundaries.

References

Fruend, Jack and Jack Jones. *Measuring and Managing Information Risk: A FAIR Approach*. Oxford: Butterworth-Heinemann, 2015.

Gemmail, Rafiq. "Trunk Based Development as a Cornerstone for Continuous Delivery." InfoQ.com. April 22, 2018. https://www.infoq.com/news/2018/04/trunk-based-development/

"Hygieia." GitHub.com. Accessed August 1, 2019. https://github.com/Hygieia/Hygieia

Pal, Tapabrata. "Focusing on the DevOps Pipeline." Medium.com. May 16, 2018. https://medium.com/capital-one-tech/focusing-on-the-devops-pipeline-topo-pal-833d15edf0bd.

Simon, Fred, Yoav Landman, Baruch Sadogursky. *Liquid Software: How to Achieve Trusted Continuous Updates in the DevOps World*. Sunnyvale: JFrog Ltd., 2018.

Sonatype. "2019 State of the Software Supply Chain." Fulton, MD: Sonatype, 2019. https://www.sonatype.com/hubfs/SSC/2019%20SSC/SON_SSSC-Report-2019_jun16-DRAFT.pdf

Notes

1. Freund and Jones, *Measuring and Managing Information Risk*, 351.

2. Freund and Jones, *Measuring and Managing Information Risk*, 351.

3. Freund and Jones, *Measuring and Managing Information Risk*, 351.

4. Pal, “Focusing on the DevOps Pipeline.”

5. Pal, “Focusing on the DevOps Pipeline.”

6. Pal, “Focusing on the DevOps Pipeline.”

7. Pal, “Focusing on the DevOps Pipeline.”

A Special Thank You to Our Sponsor

Our mission for the Forum is to bring together technology leaders across many industries and facilitate a dialogue that solves problems and overcomes obstacles in the DevOps movement. For three days at this private event, we gather 50 of the best thinkers and doers in the DevOps space to tackle the community's toughest challenges. We ask these thought leaders to collaborate and generate a piece of guidance with their best solutions to the challenges.

We would like to thank all of our attendees and our friends at XebiaLabs for helping to make this year's Forum a huge success.