

Learnable

Learnable Secure SDLC

How we build, review, deploy, and operate our software: the controls we apply at each stage of delivery, the risks they address, and the evidence we produce for each one.

OWNER	Learnable
EXPORTED	29 July 2026
SOURCE	https://sdlc.learnable.team/

This document is generated from the site above, which is built from the `main` branch of the process repository. The site is the authoritative copy; a printed or downloaded PDF is a snapshot of it.

Contents

1	Introduction	3
2	Background	6
2.1	Why do you need a process?	6
2.2	DevOps Compliance Culture	8
3	Software Process	11
3.1	Exception Register	12
3.2	Risk Register	13
3.3	Training	14
3.4	Requirements	15
3.4.1	Secure Build	15
3.4.1.1	Artifact Binary Provenance	15
3.4.1.2	Version Control	17
3.4.1.3	Defined Toolchain	19
3.4.1.4	Dependency Management	20
3.4.2	Secure Process	21
3.4.2.1	Code Review	21
3.4.2.2	Quality Assurance	22
3.4.2.3	Security Vulnerability Scanning	23
3.4.2.4	Deployment Approvals	24
3.4.3	Secure Runtime	25
3.4.3.1	Change Records	25
3.4.3.2	Deployment Controls	26
3.4.3.3	Secrets Management	27
3.4.3.4	Service ownership	28
3.4.3.5	Workload Monitoring	29

1

Introduction

The Learnable Secure SDLC

Overview

This site is the defined Secure Software Development Lifecycle for Learnable.

It describes how we build, review, and run our software: the controls we apply at each stage of delivery, the risks those controls address, and the evidence we expect to be able to produce for each one.

It applies to everyone who writes, reviews, deploys, or operates Learnable software.

This process is documented in version control, in an internal repository. The site you are reading is built from that repository's `main` branch, so it is always the current process.

Why we define a process

Defining a software delivery process is the first step towards compliance with any regulation or standard. Compliance requires us to **define** a process, **implement** it in our daily work, and be able to **prove** we followed it.

None of these standards require manual sign-offs, change advisory board meetings, or scheduled release windows. They require a defined process and evidence that it was followed — which is something automation does better than meetings.

You can read more about the reasoning behind this approach in [Background](#).

What is in this process

The process is organised into three areas, covering the whole path from source code to running production workload.

Secure Build — how software is produced and identified

- [Artifact Binary Provenance](#)
- [Version Control](#)
- [Defined Toolchain](#)
- [Dependency Management](#)

Secure Process — how changes are reviewed and approved

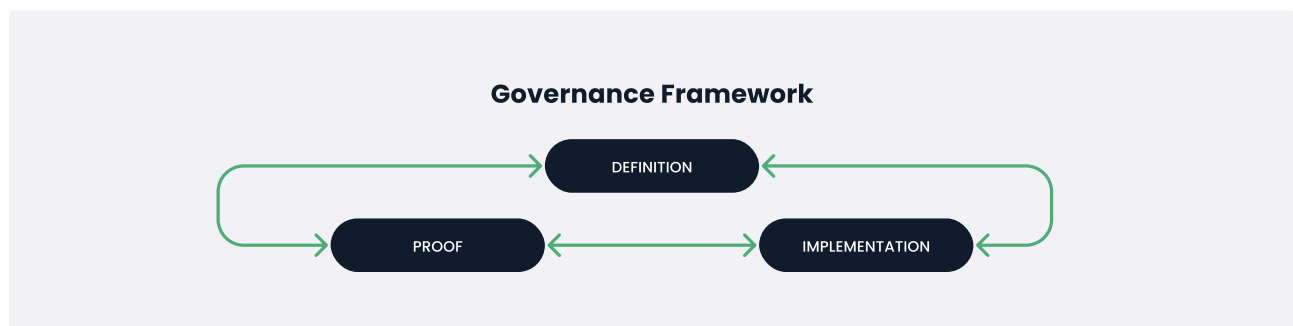
- [Code Review](#)
- [Quality Assurance](#)
- [Security Vulnerability Scanning](#)

- [Deployment Approvals](#)

Secure Runtime — how software is deployed and operated

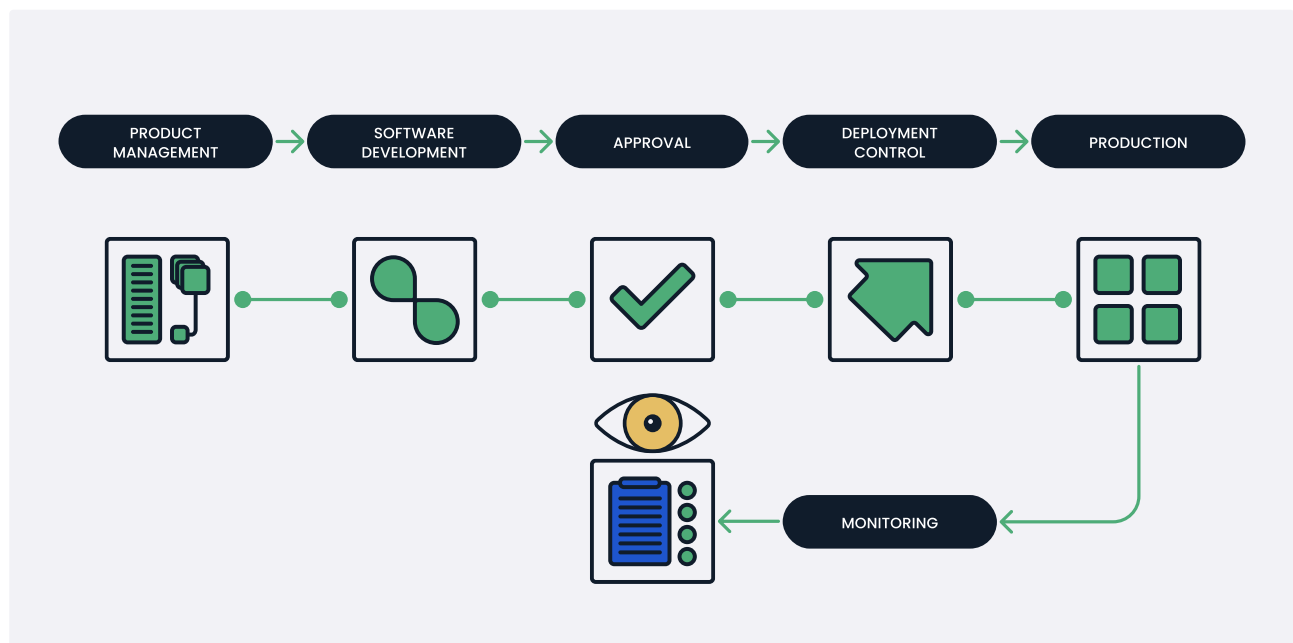
- [Change Records](#)
- [Deployment Controls](#)
- [Secrets Management](#)
- [Service Ownership](#)
- [Workload Monitoring](#)

Alongside these, the process maintains a [Risk Register](#) mapping risks to the controls that mitigate them, an [Exception Register](#) recording documented deviations and their rationale, and our [Training](#) commitments.



Scope

The scope of this framework is to secure the entire value stream of software development.



Changing this process

The process is expected to change as Learnable does. Changes follow the same route as any other change we make:

1. Open a pull request against the internal repository that holds this site.
2. Have it reviewed by someone who understands the affected control and the risk it addresses.
3. Merge to `main` — the published site rebuilds automatically.

If a team needs to deviate from a control rather than change it for everyone, record it in the [Exception Register](#).

2

Background

2.1

Why do you need a process?

Who needs to define a software process?

Regulated software companies like banks, fintechs, healthcare providers, medical device manufacturers, etc. need to comply with government legislation and/or industry standards (IEC 62304, ISO26262, NIST, FDA, etc.) before they can take products to market and sell to customers.

Many non-regulated companies also choose to opt-in to standards like SOC2 and ISO27001 to win the confidence of their customers.

To achieve compliance with any of these regulations and standards involves defining a software delivery process, implementing it, and then proving that the process is being followed.

This is why you need a software process.

Risk Management

Typical obligations in this framework include having processes around how IT systems change, and managing the risks associated with changes. Crucially, it is expected that there are comprehensive records of these changes, and the risk mitigations and internal controls performed as part of the changes.

This record of your changes should also contain evidence that your developers are following your process and performing risk controls.

Essentially, we must:

1. *Define*: Have a defined process for software development
2. *Implement*: Actually follow this process in our daily work
3. *Prove*: Have the proof that we conform to the process during audit

Governance Framework



What should be in a SSDLC process?

Generally a process should define the activities and technologies used in the development and maintenance of our software. In a regulated setting, the rationale for given controls and processes should map to the risks identified in governance.

- A risk register
- The areas of SSDCL
 - For each area:
 - A definition
 - Risks this area controls for
 - A rationale
 - Application rule (Mandatory, Optional)

How should our SSDLC be documented?

It should be defined and available on an internal website and stored in a version controlled system such as git.

Documenting Options and Exceptions

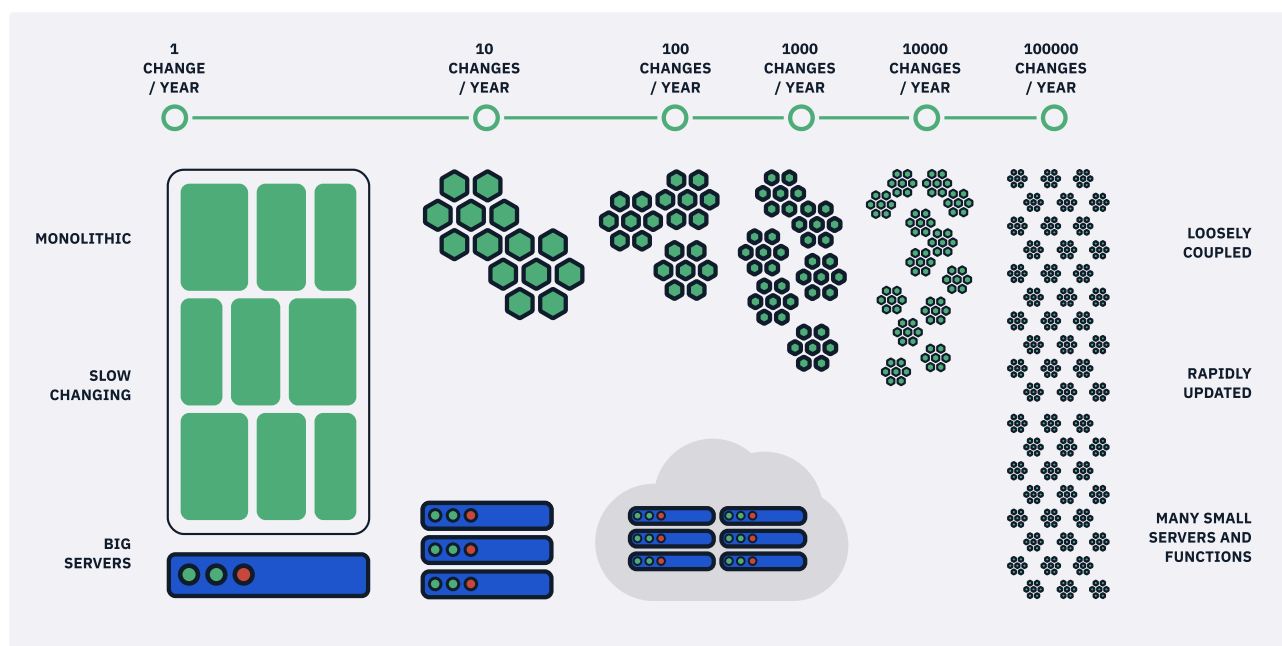
Exceptions and customizations should be defined in the exception register.

2.2

DevOps Compliance Culture

TLDR: DevOps teams who understand and conform to process with automation support outperform teams with external approvals and compliance managers.

The way we build and deliver software has changed significantly over the years. In the past, we would build systems on large, custom managed hardware. Organizations were siloed by specialty, with different departments for analysis, development, quality assurance, and operations. Software was delivered in releases, once a year, or once a quarter even. Change was rare, risky, and large.

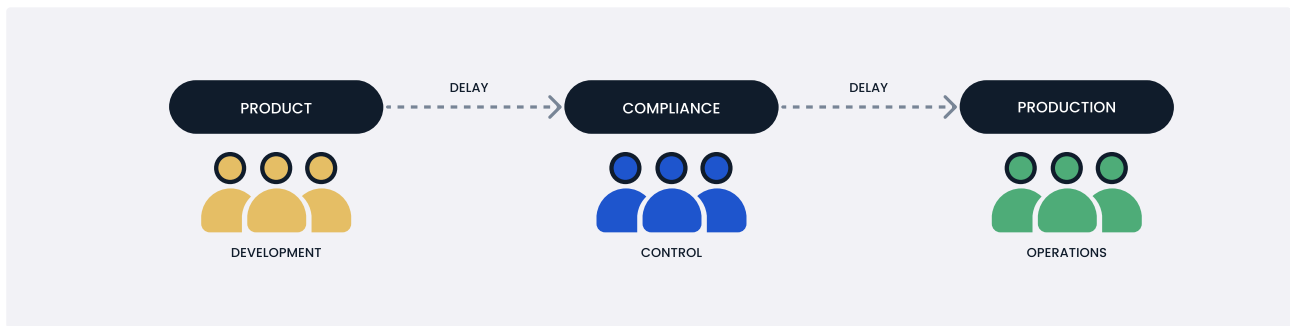


Over time, we have favoured smaller services-oriented architectures and independently deployable systems. This allowed for more dynamic and resilient designs, and to leverage Conway's Law to organize our teams around these services.

This trend continues, with the rising adoption of serverless features and functions-as-a-service available in cloud providers.

Who is responsible for compliance?

Traditionally, it was the IT department who were responsible for changing IT systems, and the volume of changes was relatively low. This also meant that the responsibility of meeting these regulatory obligations rested with the IT department.



The Compliance Bottleneck

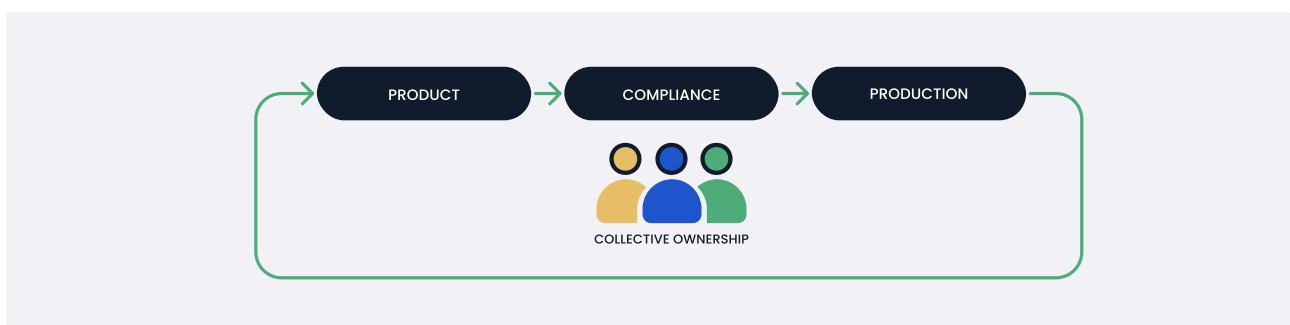
Conventional Change Management approaches in regulated industries have been IT-led, such as [ITIL practices](#), change tickets, [Change Advisory Board meetings](#), and scheduled changes. However, these processes were designed for infrequent changes and causes delays and batching changes. Somewhat counter-intuitively, these practices designed to reduce risk actually lead to increased delivery risks.

“External approvals were negatively correlated with lead time, deployment frequency, and restore time, and had no correlation with change fail rate. In short, approval by an external body (such as a manager or CAB) simply doesn’t work to increase the stability of production systems, measured by the time to restore service and change fail rate. However, it certainly slows things down. It is, in fact, worse than having no change approval process at all” ¹

To learn more about this, look into the UK Financial Conduct Authority’s analysis of over a million production changes, published in [Implementing Technology Change](#)

Building a DevOps Culture around Compliance

Modern technology organizations are organized differently. The traditional separation between Software Development, Quality Assurance, and IT Operations has been replaced by cross-functional devops teams responsible for the entire value stream of software systems. This new DevOps approach allows organizations to reduce handovers, increase collaboration, and ultimately deliver more innovation.



As organizations change, the supporting technology approaches also change. Companies are adopting metered cloud infrastructure as a service, automated build, test and security tooling. With the support of this automated devops delivery, high performing teams can deliver changes 973 times² more frequently than traditional teams. If we need to be compliant, but external approvals and management don't work, what should we do?

Continuous Compliance

This vast increase in the number of changes needs new and better approaches to change management and software process compliance. To support the efforts of devops teams, we need to move away **from manual gated checks** towards **continuous automated checks**.

Need	Traditional Compliance	Continuous Compliance
Process conformance	Checklists	Risk controls as code
Change Management	Change Tickets	Self documenting changes
Governance	Audits	Compliance monitoring

In this way, we can not only achieve more flow in our work, but actually get better compliance.

We have seen in practice that adoption of continuous compliance methodology lead to the similar benefits that test automation brings.

Adopting test automation the need for testers does not go away, merely the monotonous repetitive work. This frees testers to work at higher-level exploratory testing.

In the same way, Continuous Compliance does not mean the end of compliance work, but rather enables compliance officers to work on higher value compliance investigation.

Conclusions: the Advantage of DevOps Compliance Cultures

The devops principles around culture, automation, lean, measurement and sharing can help teams increase compliance activities while reducing delays and waste. Overall the benefits from adopting a DevOps Compliance Culture are:

- **Culture:** creates agency for teams to lead risk management responsibilities
- **Automation:** drives higher compliance conformance and reduces waste
- **Lean:** results in continuous improvement in risk control posture
- **Measurement:** ensures compliance becomes data-driven
- **Sharing:** makes compliance work visible to empower compliance and security functions

1. Forsgren PhD, Nicole. Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations . IT Revolution Press. Kindle Edition. ↩

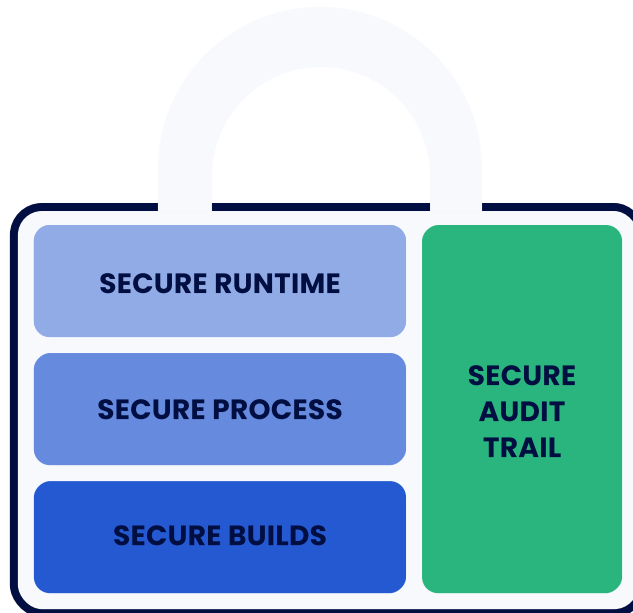
2. <https://cloud.google.com/devops/state-of-devops> ↩

3

Software Process

Introduction

This is the Secure Software Development Lifecycle for Learnable



The DevSecOps Framework defines the capabilities in a secure DevOps landscape.

The DevSecOps Framework provides:

- An vendor agnostic approach
- A holistic view of managing insider threat
- A clear roadmap for a security-based devops implementation

3.1

Exception Register

Where a team cannot meet a control defined in this process, the deviation is recorded as an exception, along with the rationale for it. Recording an exception is always preferable to quietly diverging from the process.

The current list of exceptions is maintained in the **team wiki**, which is the authoritative record. Anyone with Learnable access can find it there.

No list and no link appear on this page. This site is published publicly, whereas the register itself is internal.

3.2

Risk Register

The risk register is the list of risks this process exists to control, together with their severity, ownership, and current status.

It is maintained in **Linear**, which is the authoritative record. Anyone with Learnable access can find it there.

No list and no link appear on this page. This site is published publicly, whereas the register itself is internal.

Each control described in this process carries a rationale explaining the risk it addresses, so the reasoning behind a control stays with the control itself. The register is where those risks are tracked and reviewed over time.

3.3

Training

The team will annually go through the [OWASP top 10 security risks](#) and discuss them. The date of the session and its participants will be recorded in a compliance system of record. For new employees the OWASP top 10 will be done together with one of the other team members.

3.4

Requirements

3.4.1

Secure Build

3.4.1.1

Artifact Binary Provenance

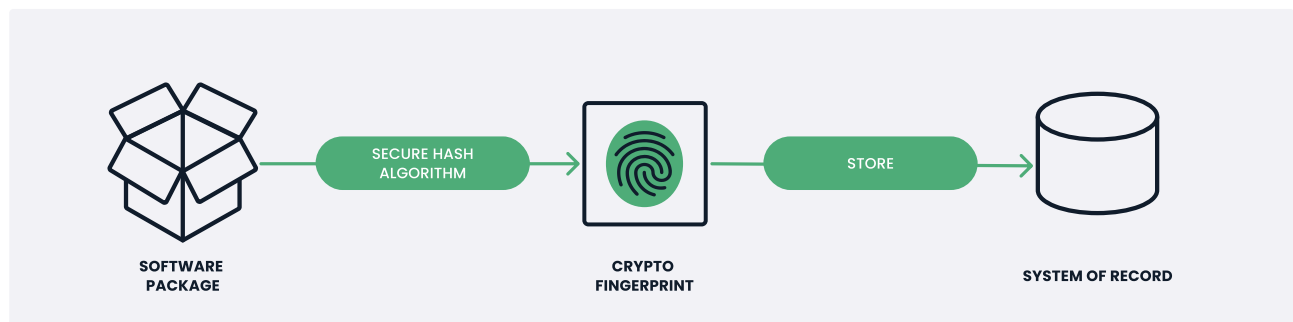
TLDR: Every software running production has known provenance

Rationale: High security environment require a tamper-proof identity scheme. The use of Content Addressable Storage mechanisms ensures that if software changes it will have a different identity.

Background

To define software identity, you use the cryptographic hash of the software itself. We use the SHA256 digest of the software binary.

This means that if a single byte in the software changes it will have a different identity.



What evidence to record

- The SHA256 of the artifact (docker image, zip file, ...)
- A human readable name
- The git commit that produced it
- The git repository state (clean, unstaged files, ...)
- A url to the build log
- The build environment information
- The software bill of materials

Alternative identification of artifacts

It can be helpful to use human-friendly identities in CI displays, filenames, and docker image tags. Some examples could be semver, or git source commits.

These are very useful ways for humans to navigate identity through version control and CI systems. However, since they are fallible, they cannot be used to identify software in the security and compliance areas.

Use labels for humans and SHAs for machines.

Further reading

- [SLSA — Supply-chain Levels for Software Artifacts](#)
- [CISA/NTIA — The Minimum Elements For a Software Bill of Materials](#)

3.4.1.2

Version Control

TLDR: Every change to the source is tracked in a version control system

Rationale: Version control allows us to track and manage changes to our software code. As a traceability system, it provides a means to understand how our software changes, who changes it, and why it was changed.

Background

We use git to manage versioning for software development source code. For repository hosting and user management we use github.

Branching Strategies

Every service will follow one of the following branching strategies:

1. Feature Branch + Pull Request, or:
2. Production Branches

These are described below.

1. Feature Branch + Pull Request

This branching strategy uses a combination of feature branches with pull requests.



- Main branch is protected
- Pull requests must be approved before merge to the main branch.
- We use pull requests to enforce and document our code review process. You can read more about it here: [Code Review Process](#)
- Pull request merges should create merge or squash commits. (no fast-forward)

Merges to the main should either be merge commits or squash commits... i.e. no fast-forward merges. This allows us to atomically back out merges should we need to.

2. Production Deployment Branch

The Production Deployment branch is an alternative to the feature-branch/pull-request strategy.

This allows a model similar to trunk-based-development, where code reviews are implemented in the merge to production.

- Production branch is protected
- Pull requests must be approved before merge to production.
- We use pull requests to enforce and document our code review process. You can read more about it here: [Code Review Process](#)
- Pull request merges will fast-forward. This means the production branch will always “point” to a commit on the main branch

Protected Deployment Branches

To ensure compliance to the code review process, we protect the branch we deploy from (main or production) with the following requirements:

- Merges require at least one approval (two if the strategy is Production Deployment Branch)
- Builds and tests run successfully
- No unresolved merge checks

3.4.1.3

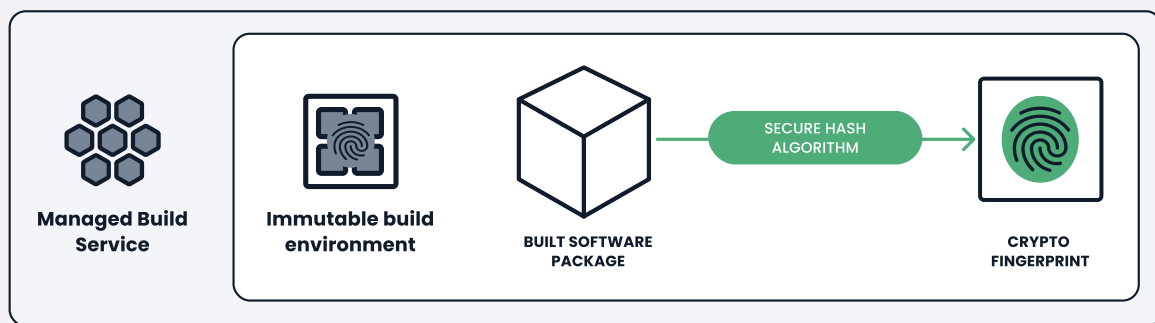
Defined Toolchain

TLDR: Build environments must be defined securely and auditable

Rationale: A secure build environment is the foundation for a mitigating software supply chain attacks. Build environments defined as code protect against interference that can happen in the build and distribution processes.

Background

Builds that are scripted, ran in an ephemeral and controlled build environment are more resilient against supply chain attacks. If at all possible, we recommend teams use immutable docker images to define the build environment. This enables auditing of the build environment, as well as security scanning and version control.



You can learn more about build security levels defined in the [slsa specification](#).

3.4.1.4

Dependency Management

TLDR: Every dependency is defined securely, managed, and auditable

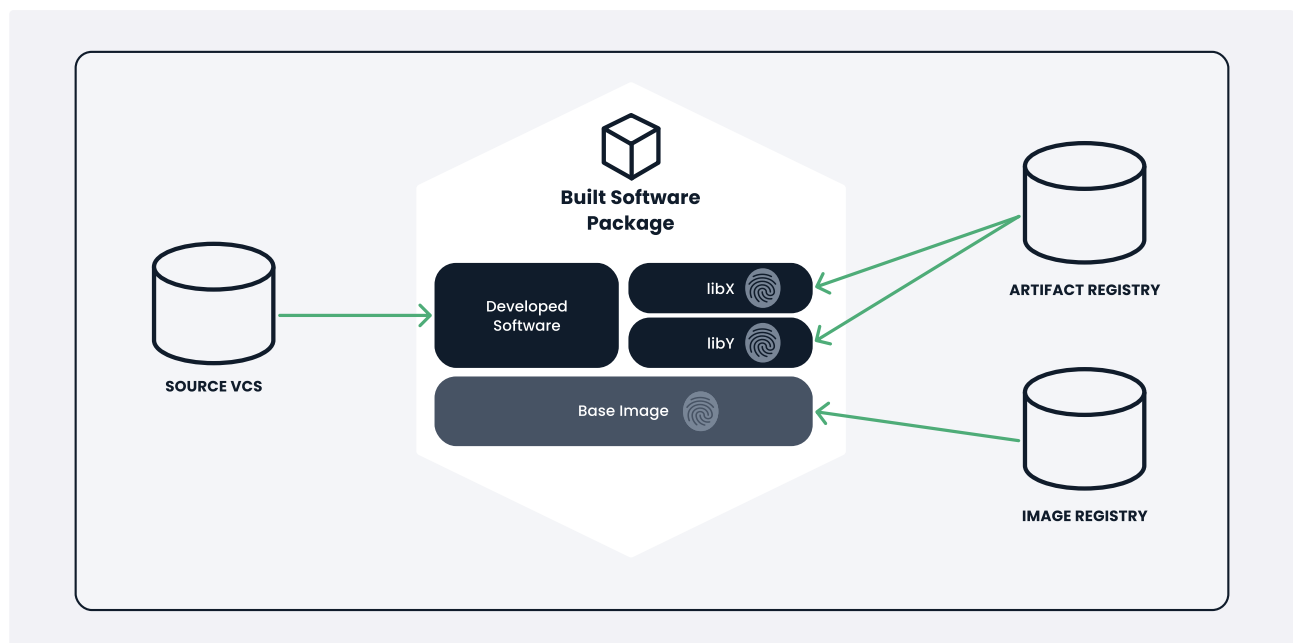
Rationale: Inputs to the build process can introduce security and quality issues, and as such must be defined, controlled, and transparent as part of the software development lifecycle.

Background

Key points:

- You must have control over what dependencies are packaged in your software
- All dependencies must comply with licensing requirements
- Must only use software with licences agreed by Learnable

Dependencies can include docker base images, 3rd-party libraries, and other source code.



During build, these inputs to the build package can be recorded as the software bill-of-materials while recording [binary provenance](#)

3.4.2

Secure Process

3.4.2.1

Code Review

TLDR: Code review is performed on all software changes

Rationale: Peer review is an essential mitigation against insider threats, as well as a means of improving knowledge sharing and quality.

Background

We use pull requests to document code reviews. The pull request description should contain key information of the change, as well as any relevant information. At a minimum, code reviews should be performed by someone capable of understanding the change and its associated risks.

Important considerations we make before approving a Pull Request:

- Security concerns: is this change secure?
- Quality: is this maintainable?
- Verification: Does this require manual testing? Has it been performed?



Code Review Anti-patterns

A common anti-pattern when using pull requests is waiting for review. In an ideal situation, the lead time for review should approach 0. If there is any delay on integration, it can lead to people batching larger and larger changes. This causes larger pull requests, more delays, poorer code reviews, and ultimately more risks.

To avoid this, we recommend pair- or ensemble-programming: a practice where more than one person works together to complete tasks. This way, as soon as one developer creates the pull request, another person can sign off immediately.

Note: the reviewer should not be the person who pushes the last commit on the branch.

3.4.2.2

Quality Assurance

TLDR: Functionality of software is assured prior to production

Rationale: Every change has the potential to introduce regressions in functionality. By testing our software prior to deployment we manage the risk of production issues.

Background

If it matters that a software system functions according to expectations, we must test and qualify our software prior to production. The level of testing and qualification should correspond to the risk appetite for a particular system. Testing can be either manual or automated (however we recommend automated approaches such as unit testing). Regardless of the approach, the testing approach must be systematic and test results documented.

The benefits of automated approaches to regression testing include:

- Less manual work
- More consistent testing
- Faster lead times
- Improved knowledge sharing
- Automated test results documentation

3.4.2.3

Security Vulnerability Scanning

TLDR: Software is scanned for security vulnerabilities prior to deployment

Rationale: Many common security vulnerabilities can be detected with automated tools. By implementing tools for dependency scanning, SAST, and DAST in the pipeline we can reduce the attack surface of our software

Background



Vulnerability scanning is the automated process of proactively identifying software vulnerabilities in software. Many common security attacks such as vulnerable components and insecure coding techniques can be prevented by integrating these tools in your devops, and acting in a timely manner on issues found.

At minimum, we recommend dependency scans in the pipelines. More advanced tools such as Synk can help identify vulnerabilities post deployment, and help with remedial actions.

- Implement security scanning in the pipeline
- Act in a timely manner to security issues
- Consider security concerns in code reviews and software design

3.4.2.4

Deployment Approvals

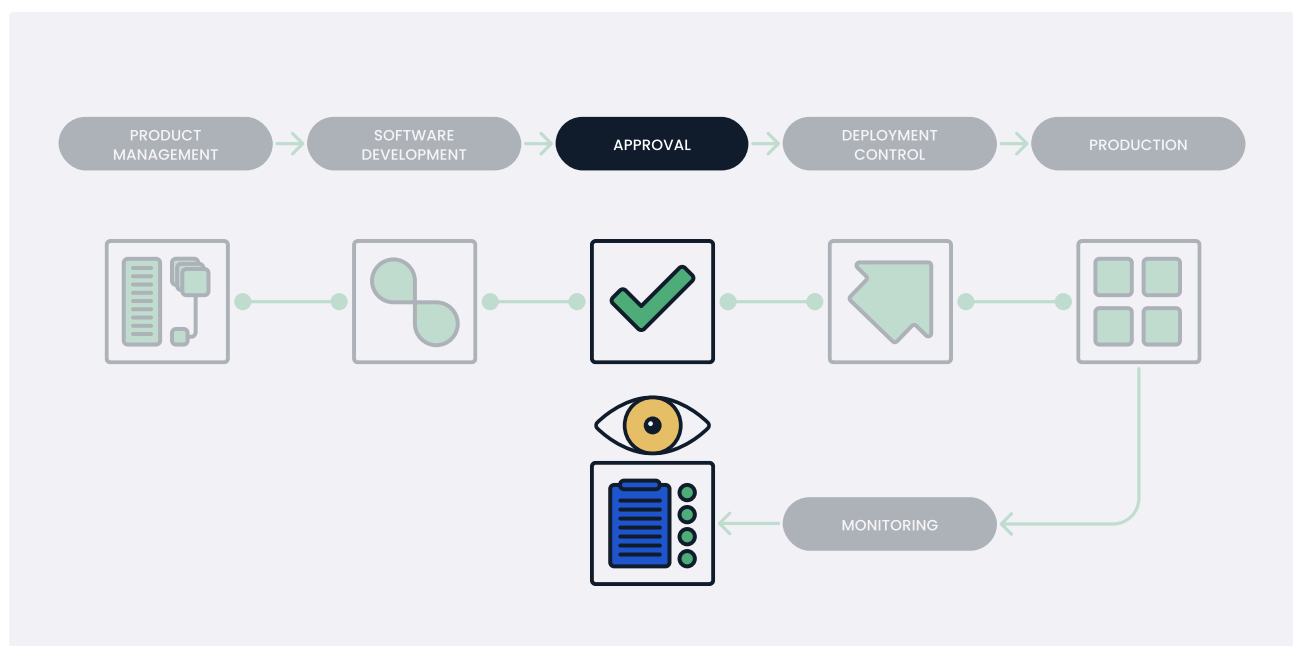
TLDR: Deployments are approved

Rationale: To meet segregation of duties requirements, all deployments to production are approved by someone other than the person making the change

Background

Segregations of duties is a common requirement in regulated or high security development environment. Put plainly, it means that a developer cannot deploy their own changes without approval from someone who both:

- Understands what is changing
- Accepts the risk of the change



Deployment approval controls form a key role in the secure software development lifecycle. Its purpose is to ensure that risks around change are managed and that change is an active decisions.

In highly sensitive software systems, more than one approver may be required.

3.4.3

Secure Runtime

3.4.3.1

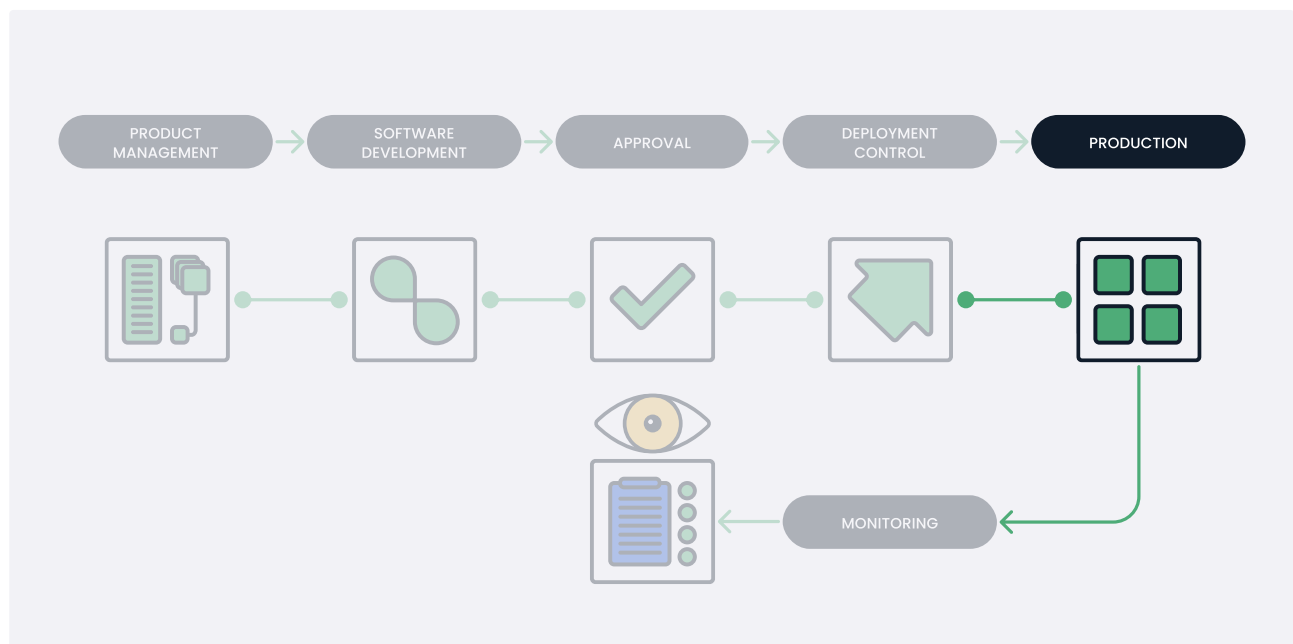
Change Records

TLDR: All systems and services maintain a record of changes

Rationale: To meet our change management requirements, all changes to production systems are recorded permanently

Background

The deployment steps in our pipelines automatically log all deployments, and we can also control that we only deploy software that is approved in the a compliance system of record audit trail.



3.4.3.2

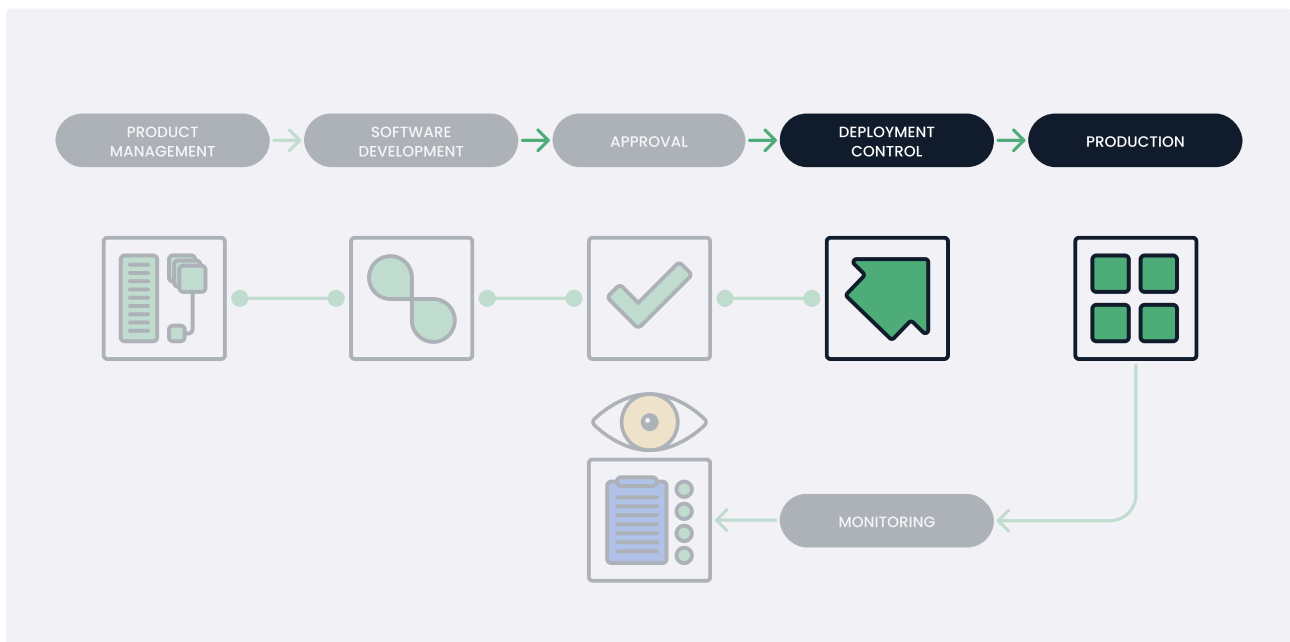
Deployment Controls

TLDR: Deployments controls are enforced in the pipeline and environments

Rationale: Ensuring only compliant, approved software deployments are made to production

Background

We use deployment controls to automatically ensure we only deploy software that has gone through our Software Development Lifecycle. This can be implemented as a gate in the pipeline, or as an admission control in the environment (ideally both).



3.4.3.3

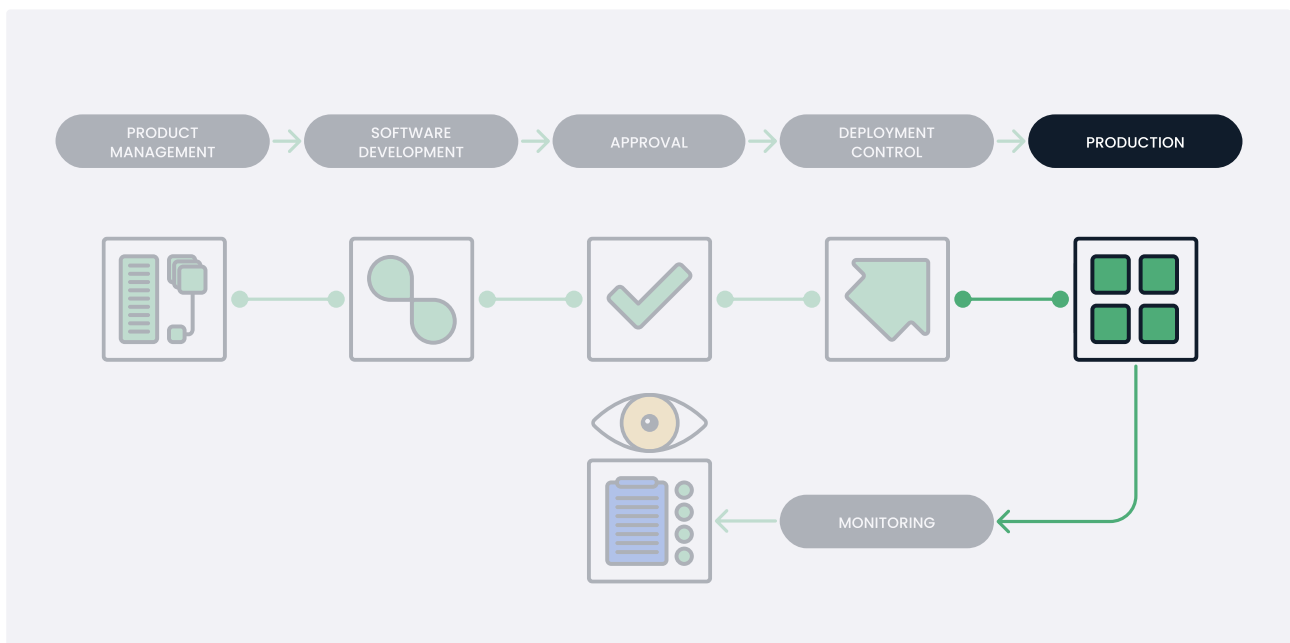
Secrets Management

TLDR: Build and runtime secrets are stored securely and documented appropriately

Rationale: Leaked secrets such as api keys, cryptography keys, identity tokens are a common attack scenario.

Background

Secrets must be stored in a secure way, and a documented in a central place. [Cryptographic failures are the second highest risk in the OWASP top ten](#) so rigor and process is essential.



3.4.3.4

Service ownership

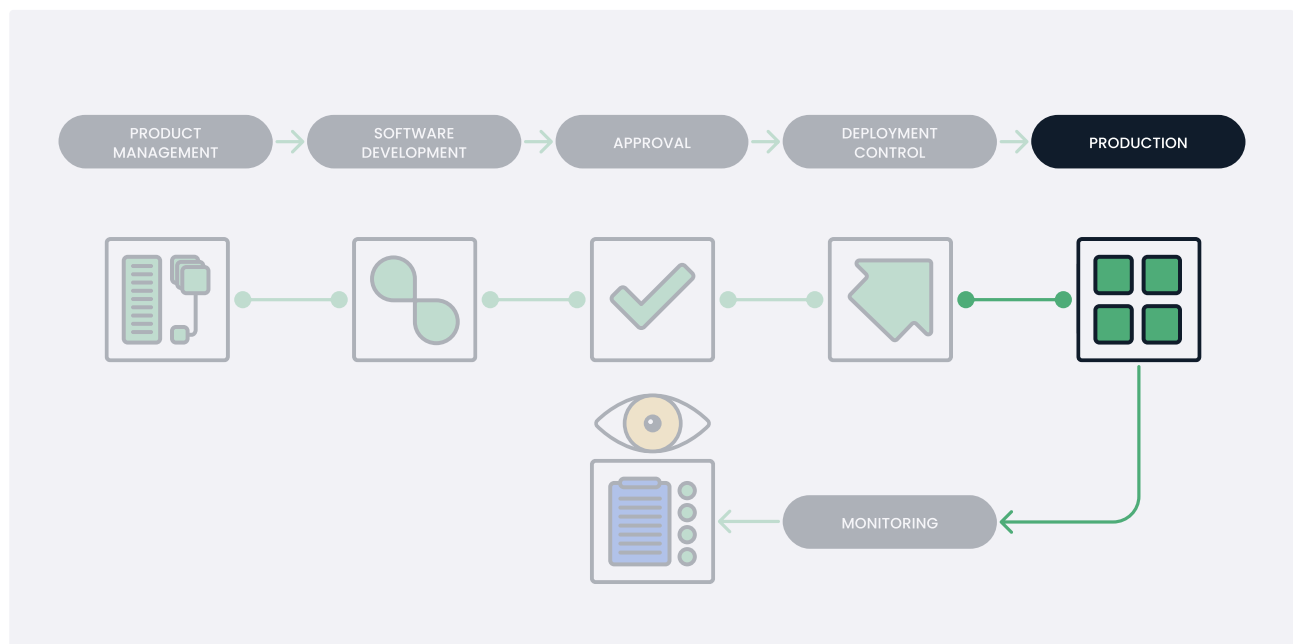
TLDR: All services running in our environments have registered ownership

Rationale: In a diverse software landscape it is essential everyone knows who is responsible for maintainance and support

Background

In any governance system, risks are managed by controls. But humans are ultimately responsible. In this context there are many reasons to keep a register of service ownership in diverse software landscapes:

- **Knowledge:** Who knows how this is supposed to work? How can I get help with this system?
- **Incident:** Alerts are firing for a service, who do I contact? What has changed lately?
- **Audit:** who is responsible that the DevOpsCTL process is followed for this service?



3.4.3.5

Workload Monitoring

TLDR: Workloads are monitored to alert if any system is noncompliant

Rationale: Real-time closed-loop compliance monitoring is a constant vigil against threats

Background

Ensuring that risks are controlled in the value stream is the first level of software process compliance. Beyond this, it is important to have a monitoring process in place to ensure that unknown or noncompliant workloads are identified in production.

