Commercial Software Distribution

< HANDBOOK />

version 3

Table of Contents

Intro

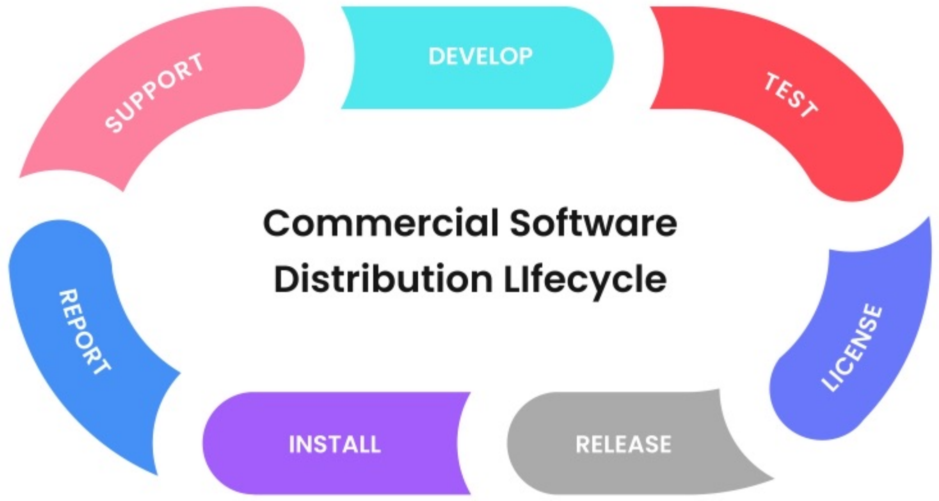
Commercial software distribution is the business process that independent software vendors (ISVs) use to enable enterprise customers to self-host a fully private instance of the vendor's application in an environment controlled by the customer.

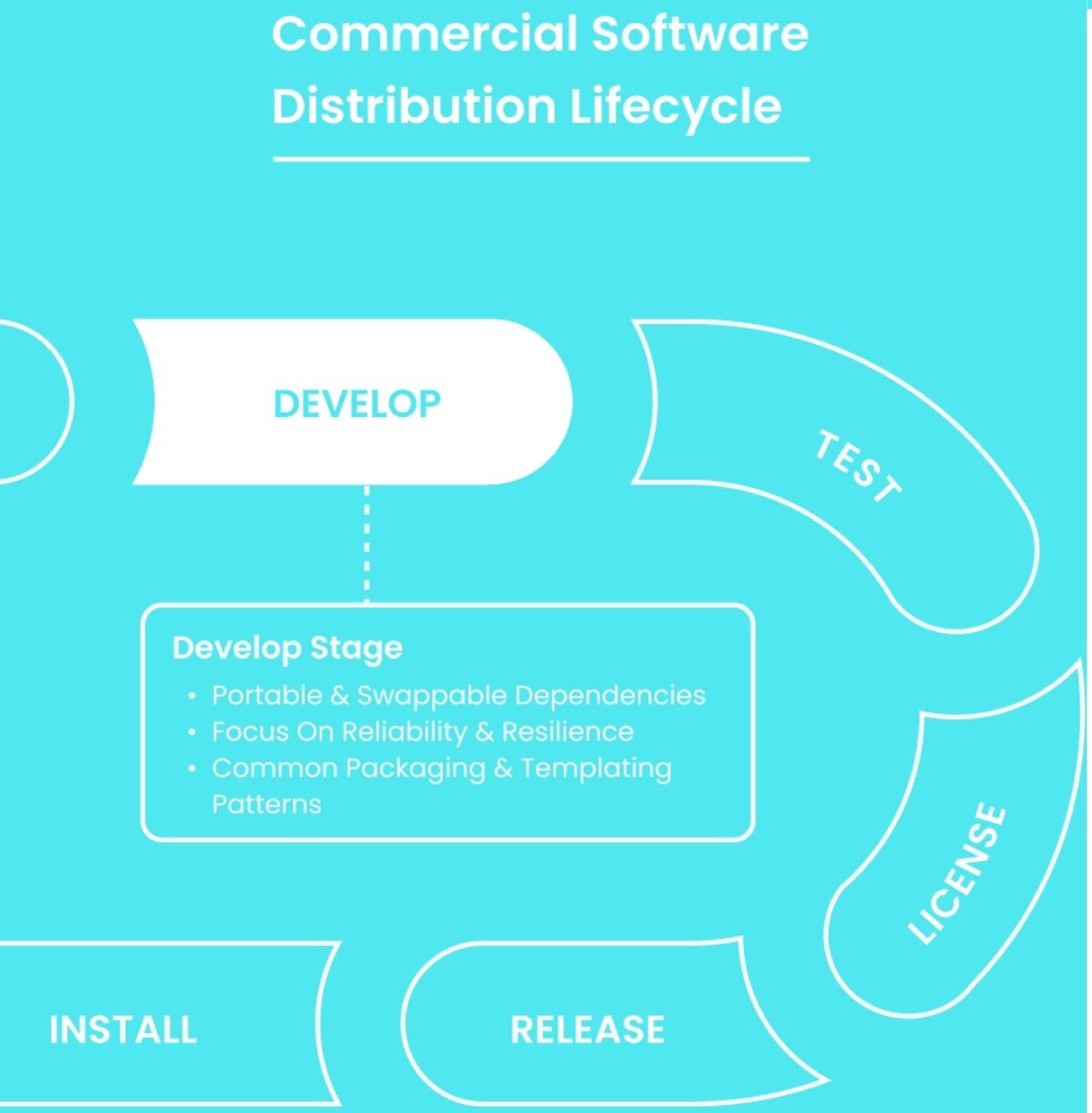
Since its inception, software distribution into self-hosted environments has changed drastically. What was once almost exclusively a process where a Solutions Engineer physically traveled to a customer's office, installed software onto dusty servers in a dark closet, and traveled back to that closet every time an upgrade was needed, is now replaced with a world of VMs, containers and private clouds. Yet, the goals for Software Distribution remain the same: deploy software, quickly and securely, into environments where the customer has complete visibility and control.

At Replicated, we've been enabling self-hosted software for nearly a decade and have worked with hundreds of software vendors as they implement a modern approach to this problem. Over this time, we've developed a unique expertise in recognizing and implementing the key steps to distributing software well. Now, we are sharing what we've learned with you and empowering every software vendor to transform how their software is distributed.

To this end, we've developed the Commercial Software Distribution Lifecycle. Developed through years of trial and error, thousands of conversations with early-stage startups and Fortune 500 companies, and pulling from the expertise of our staff, the Commercial Software Distribution Lifecycle represents the stages that are essential for every company that wants to deliver their software securely and reliably to customer controlled environments.

This lifecycle was inspired by the DevOps lifecycle and the SDLC, but it focuses on the unique things that must be done to successfully distribute third party, commercial software to tens, hundreds, or thousands of enterprise customers. The phases are:




Commercial Software Distribution Lifecycle
DEVELOP
TEST
LICENSE
RELEASE
INSTALL
Develop Stage
• Portable & Swappable Dependencies
• Focus On Reliability & Resilience
• Common Packaging & Templating Patterns

Develop

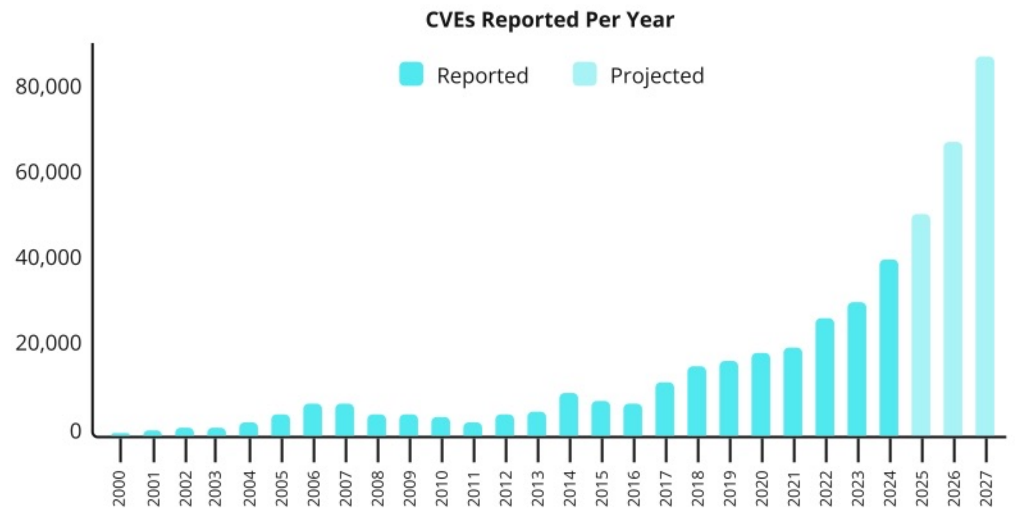
Develop refers to the technical decisions made by software vendors when developing software to be consumed by enterprise customers in self-hosted environments. To provide a seamless experience for customers, software vendors must consider how an application will be distributed while developing it—not after the fact. This includes considerations during the design, architecture, and packaging of the application. When an application is developed well, customers can install when and where they want, with confidence in the application's security posture. They aren't limited by the application's architecture and are able to bring their own requirements and tooling when needed. In other words, the application is developed in such a way that the enterprise customer can be successful, whether this is the first self-hosted application they are deploying or the 100th.

The following sections describe key considerations for vendors when developing modern commercial software for on-prem deployment, including the application's security, resilience, use of portable and optional components, and use of open standards.

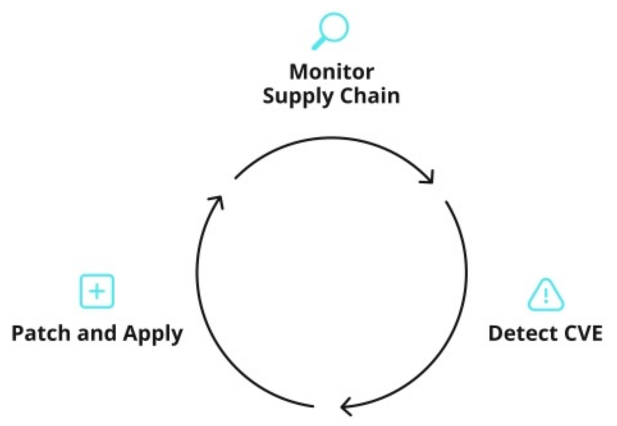
Security

One of the most important aspects of distributing commercial software is that all container images used by the application, including dependencies, are free of Common Vulnerabilities and Exposures (CVEs).

In today's commercial software industry of complex software supply chains, fast-paced development cycles, and large attack surfaces, keeping up with the near-constant flow of CVEs can feel overwhelming. Additionally, given the high financial, social, and real-life consequences of a security breach, many enterprises require up-front proof that software is free of CVEs and that any new CVEs are detected and patched as soon as possible.



In this modern software development ecosystem, vendors can no longer default to scanning for CVEs on a regular quarterly or monthly schedule. Instead, CVE continuous patching needs to be included in the software development and build processes. With a continuous patching approach, automation is set up to continuously monitor the application's supply chain, detect CVEs, and then test and apply patches as soon as they are made available. Third-party platforms that offer a library of vulnerability-free container image builds (like Chainguard Images, Replicated SecureBuild, and Docker Hardened Images) can also help by making it easier for vendors to develop their applications using security-hardened images with all the latest CVE patches. This shifts CVE patching left by placing the emphasis on producing software with the latest, CVE-free images during the build process rather than on patching CVEs after deployment.



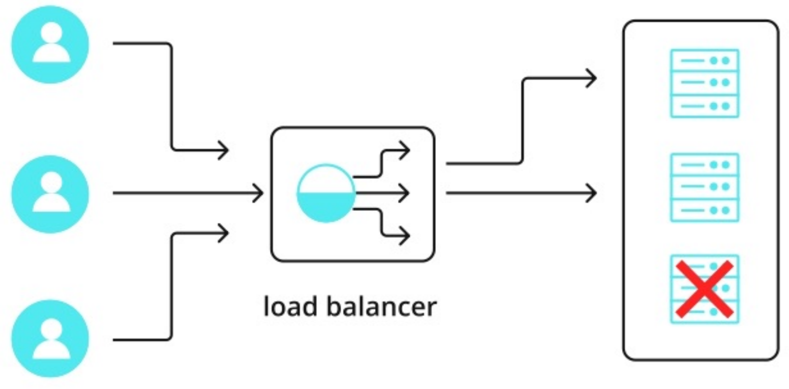
In addition to continuous CVE patching, other important security considerations to account for during the Develop phase include:

- The application follows the principle of least privilege to minimize the privileges required from the enterprise customer's cluster, including avoiding tools that require escalated permissions. For example, don't expect to be able to deploy operators into existing clusters as RBAC rules might not allow it.

- The application is developed and tested with an understanding of the different network settings that enterprises will require. For example, the vast majority of enterprise customers are not going to give inbound access to their instance of the application. Others will allow outbound access, either directly or through a proxy. The most security-conscious customers will deploy in air-gapped environments without any outbound access to the internet. These network settings can complicate installation, updates, reporting, and application operations if not thoroughly tested.

Resilience

The application is resilient. It comes back on failures and is also built to avoid externally-facing failures. For example, leveraging cloud-native Kubernetes best practices for packaging will provide you with a dependable solution for resilience and high availability.

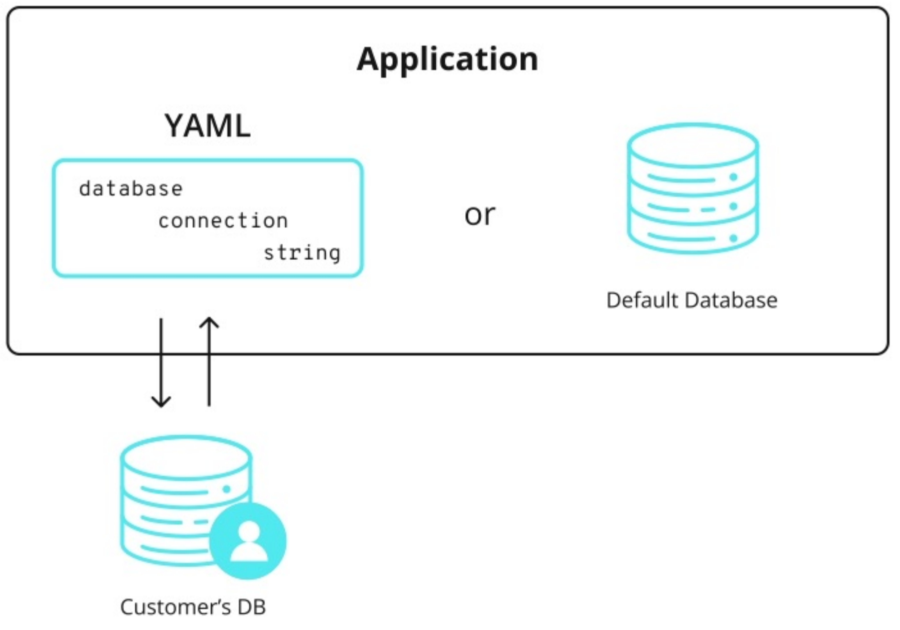


Additionally, because availability is a key aspect of software security, ensuring that an application has high availability also helps to defend against security risks that target uptime, such as Denial of service (DoS) attacks.

Portable and Optional Components

The application uses components that are portable, so vendors and customers are not locked in to specific cloud providers or services. This makes the application more self-contained and portable. For example, using an open-source queuing solution like RabbitMQ as opposed to AWS's SQS will provide more portability. Portability is also important for disaster recovery and availability in that it allows the application to be deployed anywhere at any time if needed.

The application also allows customers the choice of supplying required services as part of the installation, as opposed to utilizing whatever services are embedded with the application. For example, enterprise customers should be able to provide a connection string to their own database rather than being required to use the embedded database option.

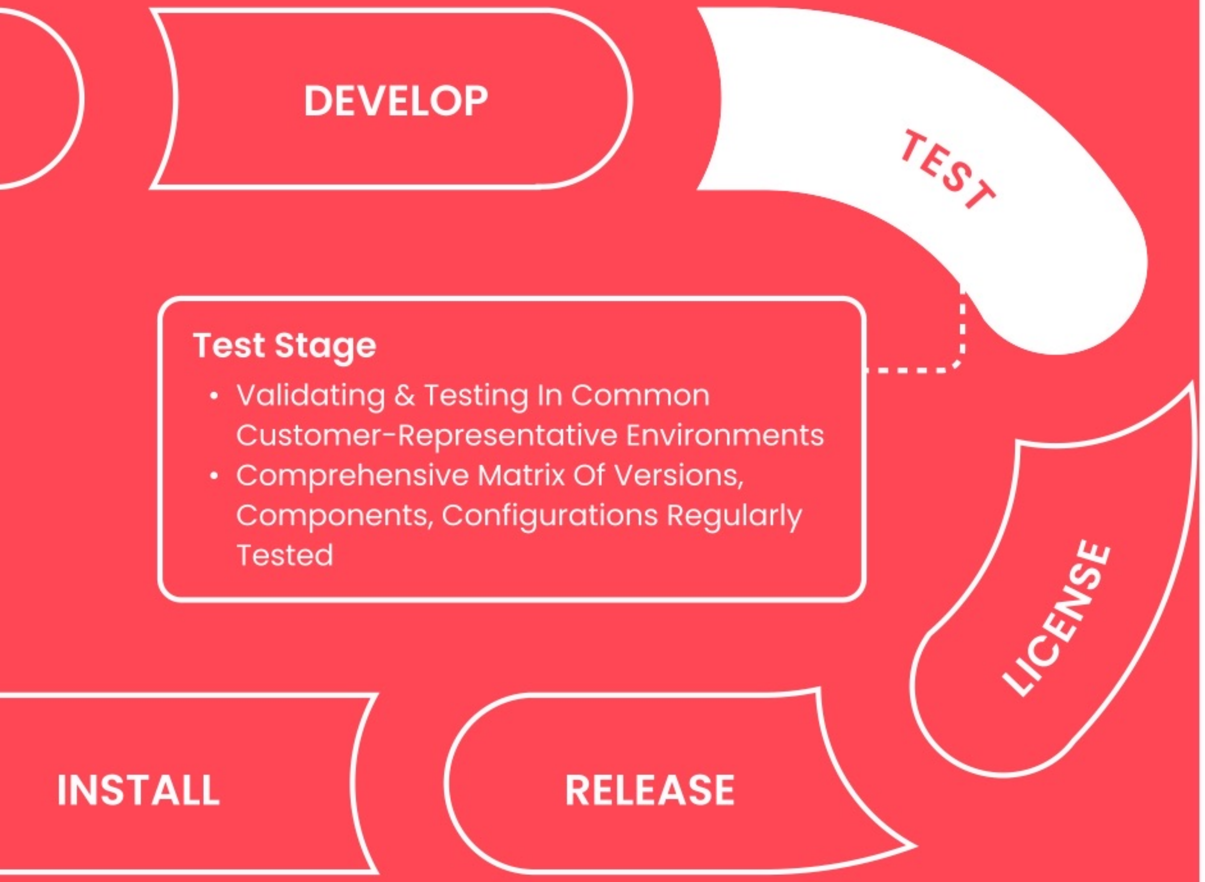


Open Standards

The application is packaged with open standards whenever possible,rather than providing custom scripts or bespoke installers. For example, most enterprise customers are familiar with Helm because it provides a consistent, reusable, and shareable packaging format. For customers who aren't familiar with Helm or Kubernetes, vendors can provide alternative installation options that treat Kubernetes as a dependency without exposing it to the customer.

Using open standards also makes it easier for an application to have interoperability with external security management or monitoring tools. Considering the wide variety of security tools that might be used by enterprises, vendors should stick to open standards rather than trying to build custom integrations.

Commercial Software Distribution Lifecycle



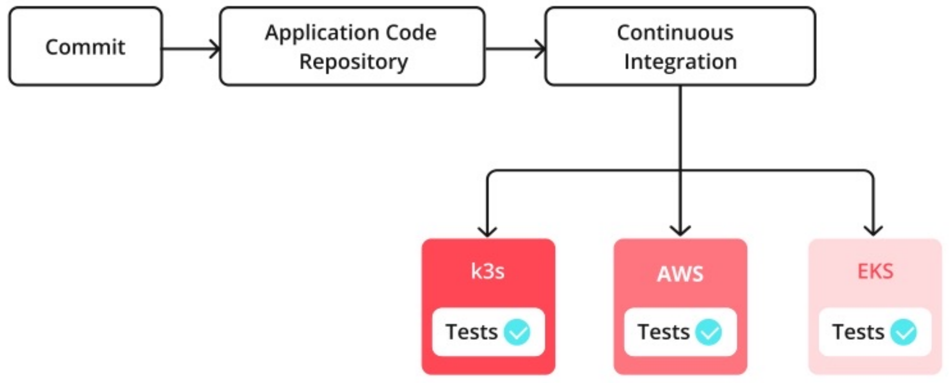
Test Stage

- Validating & Testing In Common Customer-Representative Environments
- Comprehensive Matrix Of Versions, Components, Configurations Regularly Tested

Test

Testing refers to ensuring that enterprise software can be successfully, reliably, and securely distributed to current and future customer-managed environments. Importantly, testing needs to happen before the application ever gets into the hands of customers. Not only does catching issues ahead of time help ensure the best possible experience for customers, but it's also a good way for vendors to manage costs: discovering and fixing a bug after the fact is more expensive than investing in the proper testing stack up front.

Testing self-hosted software presents a unique challenge in that both the application and the customers' environments could be the cause for a failed deployment. Different customer environments act differently, and the same application successfully installed in one customer environment could present challenges in the next. For this reason, testing self-hosted software needs to go beyond unit and integration tests to verify that the software is functional and secure regardless of the environment where it's deployed.



Software vendors should consider the following when creating a testing stack for modern on-prem software distribution:

Test in Customer-Representative Environments

Vendors can increase confidence that their application will run in any environment by testing on a variety of distributions or operating systems that are representative of their current (and future) customer's environments. For example, depending on the customer base, vendors might want to test on vanilla Kubernetes, at least one cloud-based provider (like GKE or AKS), and a more complex Kubernetes distribution like OpenShift.

When writing tests for each environment, expect different environments to act differently. For example, don't anticipate a deployment into an OpenShift environment to require the same steps as a deployment into AKS. Each customer-representative environment might require different elements to be tested.

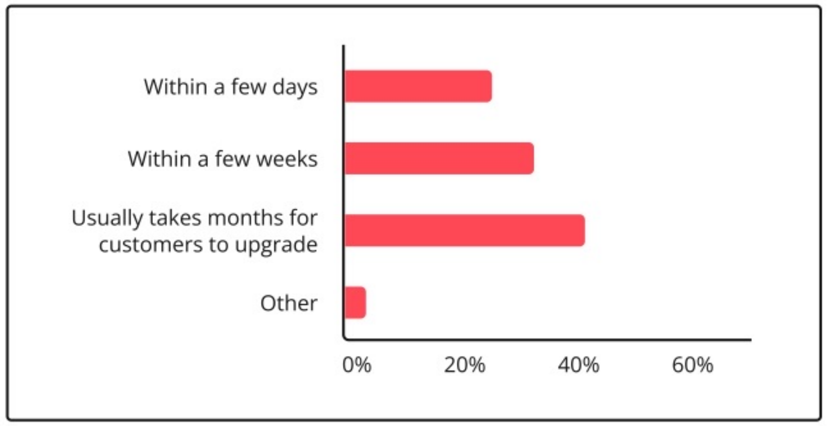
Test for Security Vulnerabilities

Validate that multiple configurations (such as different database configurations or different core services) deploy securely without exposing misconfigurations that attackers could exploit.

Testing for security vulnerabilities is especially important for self-hosted deployments because, as noted in Replicated's State of Self-Hosted Software Survey 2025, on-prem customers often take months to adopt a new release, meaning that unresolved vulnerabilities could expose these customers to risk for a longer period of time.

Question: On average, how long after the release of a new version do your customers upgrade?

Insight: Most ISVs report that customer upgrades lag weeks or months behind a new release, with over 40% saying it usually takes months. This type of delay can impact support, docs, security, new feature development, and more.

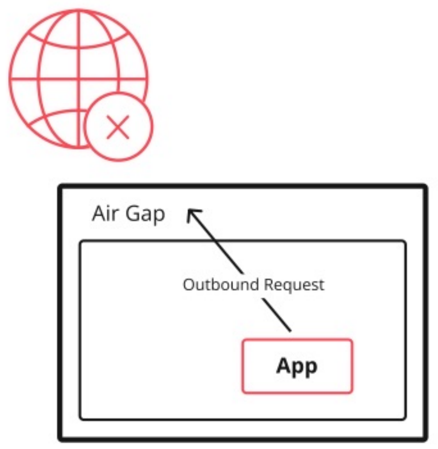


Source: The State of Self-Hosted Software 2025: A Survey by Replicated, September 2025

Test in High-Security Environments

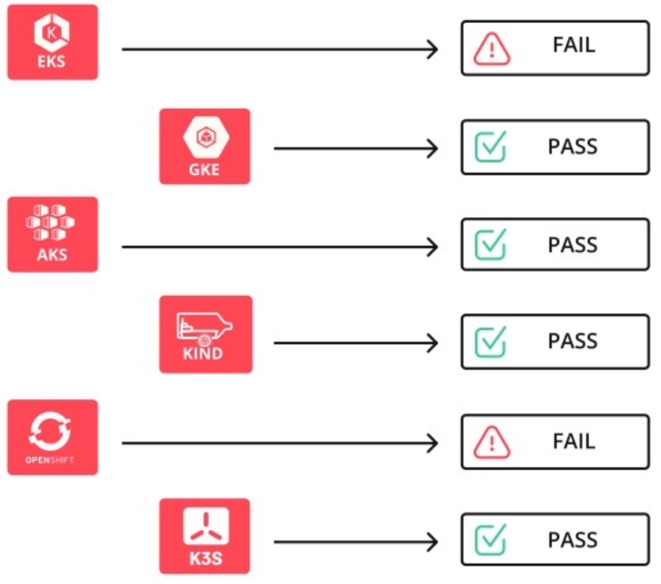
In addition to testing application releases for potential security vulnerabilities, vendors should also run tests to verify that their application can run in high-security environments, such as FIPS-compliant clusters, restricted egress and air-gapped networks, or hardened nodes.

For example, for environments that restrict outbound network requests (including air-gapped environments), add tests that identify any outbound requests made by the application and its dependencies.



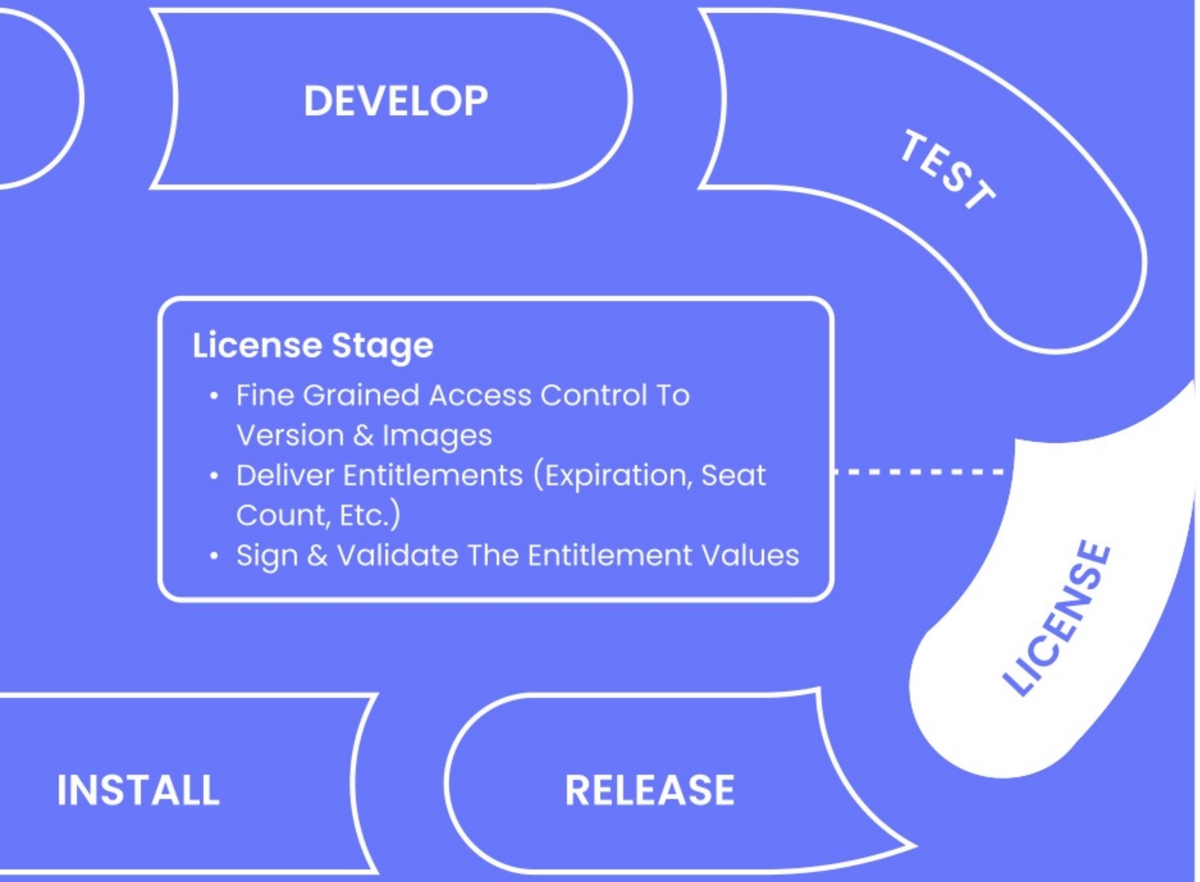
Automate Testing

Add tests to your continuous integration (CI) pipelines. This ensures that tests run automatically before each new release without requiring manual intervention from the team. Logic can also be added to CI pipelines to prevent releases from being shipped unless all tests pass.



CI pipelines should include not only functional tests but also integrate third-party security scanners. For example, vendors can integrate web vulnerability scanners to check for a variety of exploitable issues like known CVEs in web frameworks, or Dynamic Application Security Testing (DAST)-style tools that can find vulnerabilities in a running instance of an application.

Commercial Software Distribution Lifecycle



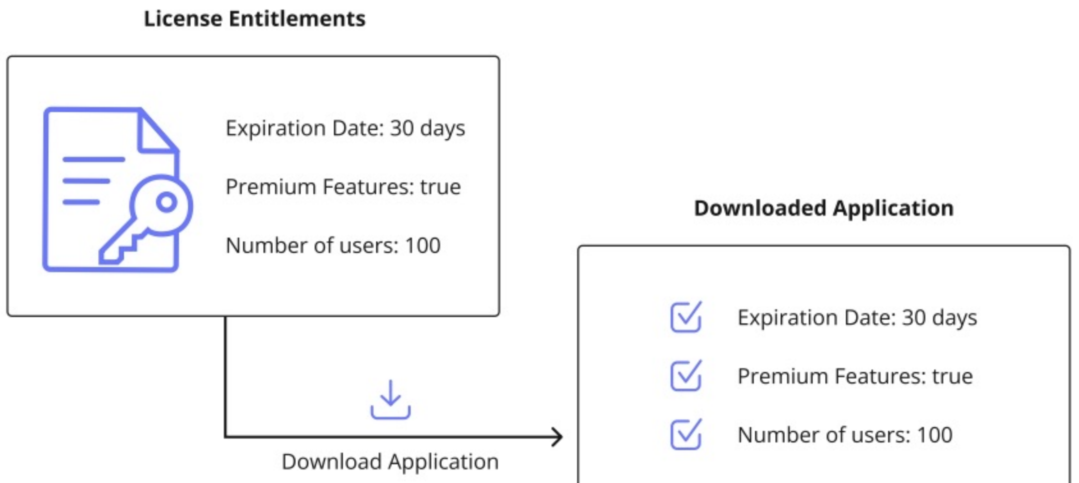
License Stage

- Fine Grained Access Control To Version & Images
- Deliver Entitlements (Expiration, Seat Count, Etc.)
- Sign & Validate The Entitlement Values

License

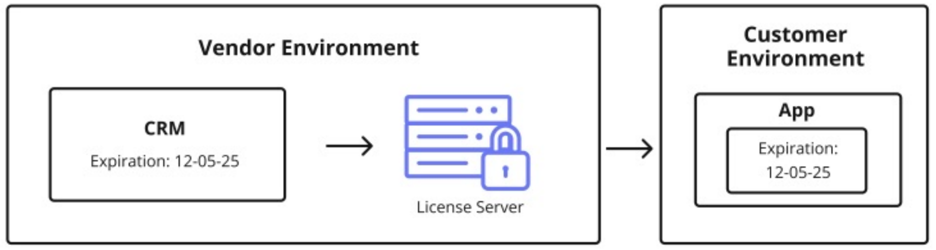
Licensing refers to securely granting access to software. Licensing codifies the agreements defined in the software contract between the vendor and the enterprise customer, and makes those agreements available to the underlying application through a license server during startup or runtime.

Licensing also provides another important data point for enterprise customers to be able to verify the integrity of the software that they are installing. For example, if the license enables the expected set of features (entitlements), then the customer can be more confident that the software is legitimate and came from the intended source. In contrast, with unlicensed software, the customer does not have access to this important data point when attempting to confirm the legitimacy of downloaded software.



Licensing is a cross-functional concern as it is important to many different teams that licenses are easy to create, update and sync to customer instances:

- For Sales teams, the license server that keeps track of users and entitlements should be integrated with internal CRM tools, such as Salesforce. This allows customer entitlements to be easily turned on and off based on changes to the software contract.



- Support teams should be able to use the license as a unique customer identifier to get visibility into insights such as the customer's entitlements and product usage.
- For Engineering teams, it is important that application logic can be used to control access to code and images so that engineers do not need to update code each time a license agreement changes.

License Entitlements

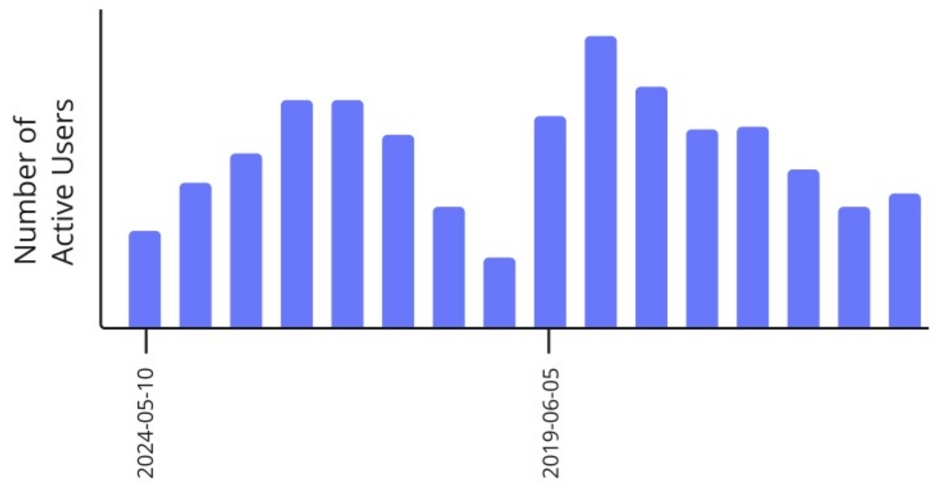
License agreements for enterprise software often include entitlements that address the following common concerns:

- Enforcing expirations of licenses, such as trial or Proof-of-Concept licenses
- Controlling feature-based and usage-based entitlements to facilitate product assortment. For example, license entitlements can determine a customer's access to a feature that is available only under a certain product plan
- Limiting the number of instances of an application that can be run by a single customer
- Other application- or customer-specific entitlements. For example, many AI applications require granular restrictions for model images to control access to sensitive data, and so it is necessary to define which images users can access. Other entitlements might include the number of users permitted, the number of nodes permitted, and so on.

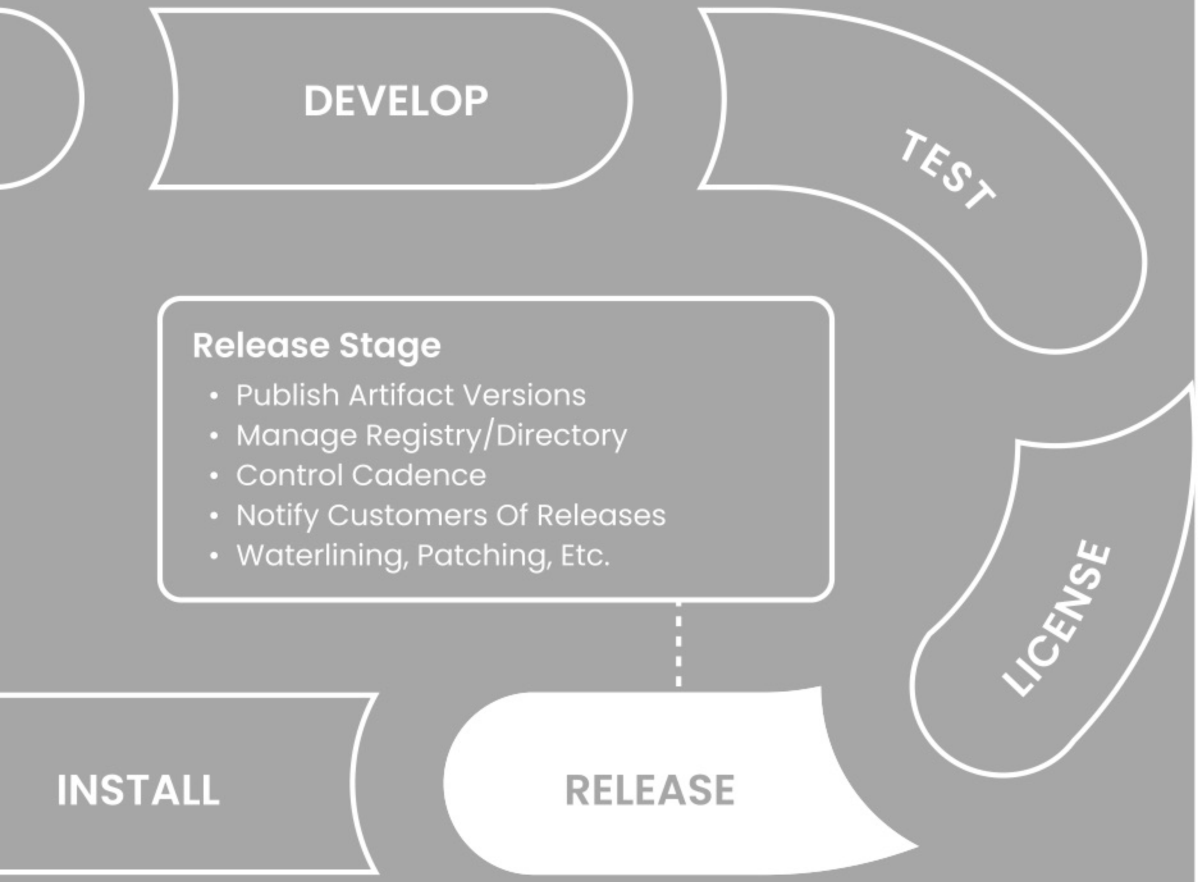
Measuring and Reporting Usage

In most cases, software vendors can confidently rely on the license agreement or contract to enforce entitlements, as enterprise customers will be wary of violating a software contract. Because of this, it is likely unnecessary to write application code that prevents certain actions or blocking usage. Instead, most software vendors will track or communicate usage that exceeds the contract and "true up" customers at renewal. One exception to this strategy is enforcing expiration dates, which can be easily extended by the vendor as needed to ensure that the enterprise customer can continue using the software.

Measuring usage surfaces relevant data to both the vendor and the customer without the negative consequence of reducing or preventing usage of the software. For vendors, it is helpful to know how customers are using the software to identify opportunities to extend or expand the agreement. For customers, understanding their own usage is valuable for avoiding violations of the contract.



Commercial Software Distribution Lifecycle


DEVELOP
TEST
LICENSE
RELEASE
INSTALL
Release Stage
• Publish Artifact Versions
• Manage Registry/Directory
• Control Cadence
• Notify Customers Of Releases
• Waterlining, Patching, Etc.

Release

Releasing refers to the process of delivering software to licensed users, ensuring that new features, improvements, and bug fixes get into the hands of the right customers at the right frequency.

Key considerations for vendors when releasing modern enterprise software include:

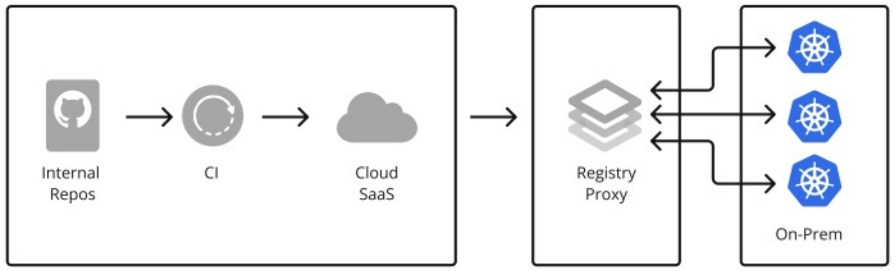
- Making application images available for customers to access securely
- Packaging and publishing cryptographically signed release artifacts in multiple formats to support different installation methods and customer environments
- Demonstrating the integrity of each release by providing software supply chain metadata like a Software Bill of Materials (SBOM) and provenance attestation, as well as publishing results of vulnerability and compliance scans
- Managing release streams for different customers, including automating workflows for production (GA) and pre-release (alpha, beta) versions
- Versioning releases with a consistent pattern so that it is easy for customers to understand backward compatibility

The following sections explain each of these considerations in greater detail.

Provide Secure Access to Images

A single release for an application contains all the artifacts required to install and run the application, such as container images or executables. When publishing a release for distribution to self-hosted environments, software vendors need to make images available to customers securely.

For online (internet-connected) environments, proxy servers can be used to grant proxy, or pull-through, access to images. Proxies work as an intermediary between the software vendor's private image registry and the enterprise customer, allowing users to access images on the vendor's private registry without exposing registry credentials. With a proxy, customers can access images using credentials determined by the software vendor, such as providing their unique license or customer ID.



For air gap environments, customers must have access to downloadable archives that contain the release images so they can push images to their own registry.

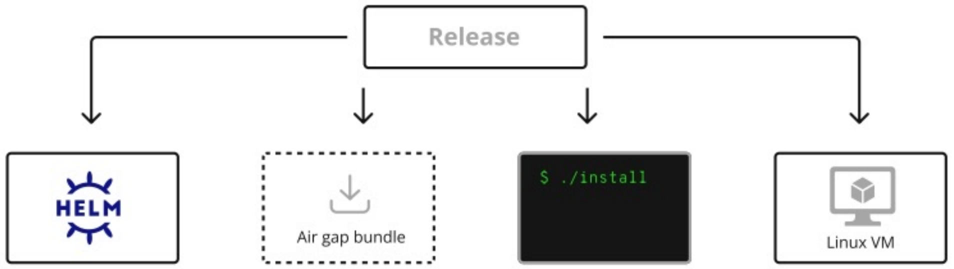
All of this activity can be tracked for auditing and reporting.

Publish Signed Release Artifacts

Releases should be made available in multiple formats to support different installation methods and customer environments. For example, while some enterprise customers with Kubernetes expertise will prefer to install in their own cluster, others will prefer to install on a virtual machine (VM) or bare metal server. Additionally, enterprises with an emphasis on security might need to deploy software in air gap environments with no outbound internet access.

Allow enterprise customers to choose the release asset they need based on their unique preferences and requirements. For example, a vendor might need to publish all of the following for a single release:

- Raw artifacts, such as Helm charts or containers
- Downloadable archives that contain the release images for air gap installations
- Installation scripts, such as scripts that install the application in Kubernetes clusters or on VMs
- Release assets that are specific to the operating system, such as unique assets for installations in Linux or Windows environments

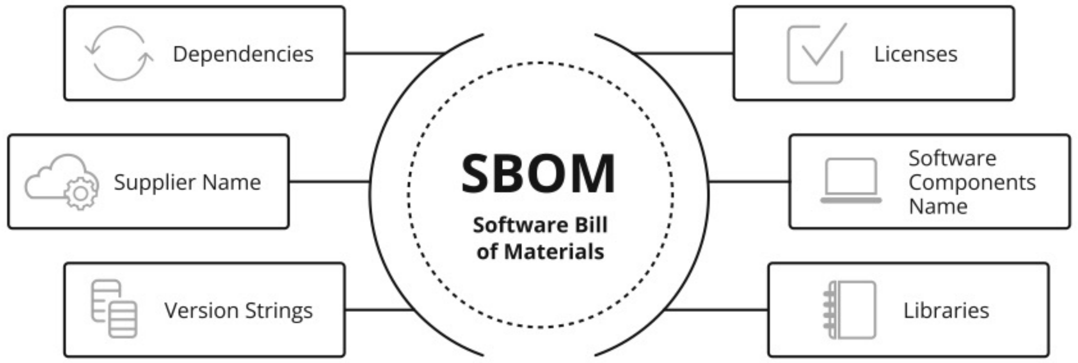


Additionally, all release artifacts should be cryptographically signed so that enterprise customers can verify the authenticity and integrity of the software before installing. Tools such as Sigstore Cosign can be used to sign and verify release artifacts automatically as part of a CI/CD pipeline.

Demonstrate Supply Chain Security

With the high volume of CVEs in modern software supply chains, enterprises are increasingly security-conscious. Vendors can convey a mature security posture to enterprise customers by proactively publishing supply chain metadata for each release, offering demonstrable proof that their software supply chain is secure, compliant, and traceable.

First, it is common for enterprise customers to require an SBOM, which provides an inventory of all components, dependencies, and metadata that make up a piece of software. Third-party tools like Syft make it easier to generate SBOMs in standardized formats as part of the release process so that vendors can then make them available to customers.



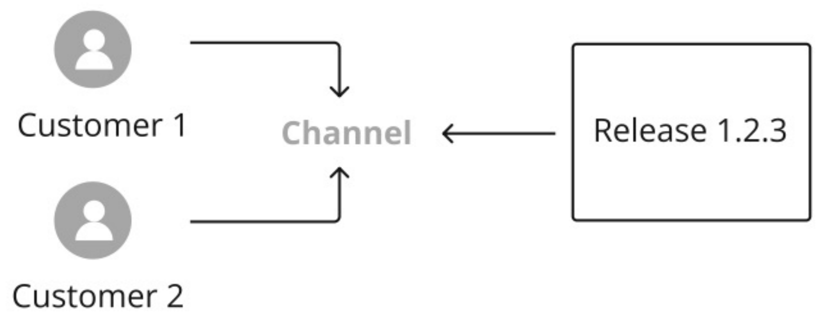
In addition to SBOMs, vendors can further demonstrate the security posture of their software supply chain by providing the following:

- Provenance attestations which include verifiable information about where, when, and how the software was produced
- The Supply-chain Levels for Software Artifacts (SLSA) security level, which is based on industry consensus security guidelines and communicates the integrity guarantees of the software
- Results of vulnerability scans to show that the software meets enterprise security and compliance requirements

Release Management

Release management is also important for ensuring that each release is made available to the right subset of users (including internal teams and customers), and that the vendor has control over the frequency that new releases are published. A successful release management practice achieves these goals with minimal maintenance burden for the vendor.

One common release management strategy is publishing releases to different channels or lanes. For example, vendors might keep separate channels for internal-only, experimental, beta, and generally available (GA) releases. Enterprise customers and internal users can then access the releases published to the channel where they are subscribed.



One common release management strategy is publishing releases to different channels or lanes. For example, vendors might keep separate channels for internal-only, experimental, beta, and generally available (GA) releases. Enterprise customers and internal users can then access the releases published to the channel where they are subscribed.

Channels can be useful as a release management tool because they allow vendors to create a logical separation between different types of releases, including those releases intended only for internal development and testing, without having to manually grant and restrict access to features or risk accidentally releasing code that could include security vulnerabilities, bugs, or sensitive information. Channels also provide flexibility in release frequency, allowing vendors to more quickly publish updates to internal or pre-release channels while maintaining a different pace for GA releases.

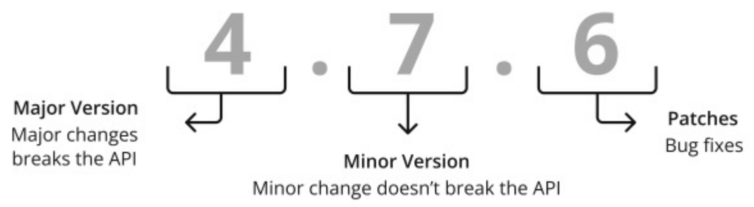
To minimize the need for manual intervention when releasing new versions, continuous integration and continuous delivery (CI/CD) pipelines should include workflows that automate release management and publishing. For example, vendors could create Github workflows that run tests, publish releases to the right channel, and notify customers subscribed to the channel that a new version is available.

Release Versioning

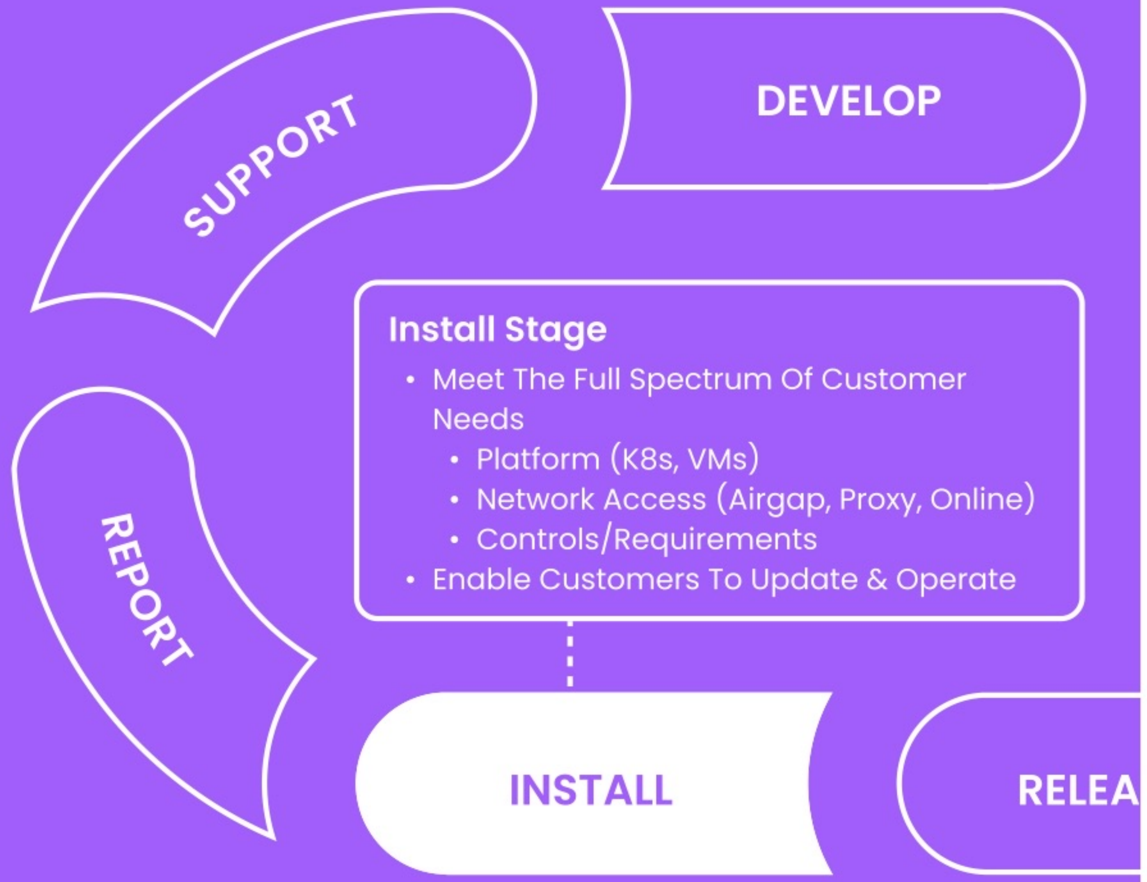
Software vendors should assign and increment version numbers for releases using a consistent pattern, such as Semantic Versioning (SemVer). SemVer is a commonly-used and recommended versioning strategy that provides implicit information about the backwards compatibility of each version, using the format MAJOR.MINOR.PATCH.

The release versioning pattern used should also dictate how build metadata and pre-release versions are indicated. For example, with SemVer, alpha or beta versions are denoted by appending a hyphen followed by the pre-release label or version number, such as 1.0.0-alpha or 1.2.3-0.0.2.

A consistent versioning pattern such as SemVer is important for modern commercial software distribution because it is common for vendors to support (and continue to release patches on) multiple different versions of their software concurrently. SemVer enables this because enterprise customers can easily understand that a new patch release is backwards compatible with the corresponding minor version, without needing to worry about breaking changes.


4 . 7 . 6
Major Version
Major changes breaks the API
Minor Version
Minor change doesn't break the API
Patches
Bug fixes

Commercial Software Distribution Lifecycle



Install Stage
- Meet The Full Spectrum Of Customer Needs
    - Platform (K8s, VMs)
    - Network Access (Airgap, Proxy, Online)
    - Controls/Requirements
- Enable Customers To Update & Operate

Install

Installing refers to the steps that enterprise customers take to securely deploy software in their environment. For modern on-prem software, the installation process varies depending on the release delivery method and the installation environment (such as internet-connected versus air-gapped, if the installation environment uses a proxy, or Kubernetes clusters versus VMs).

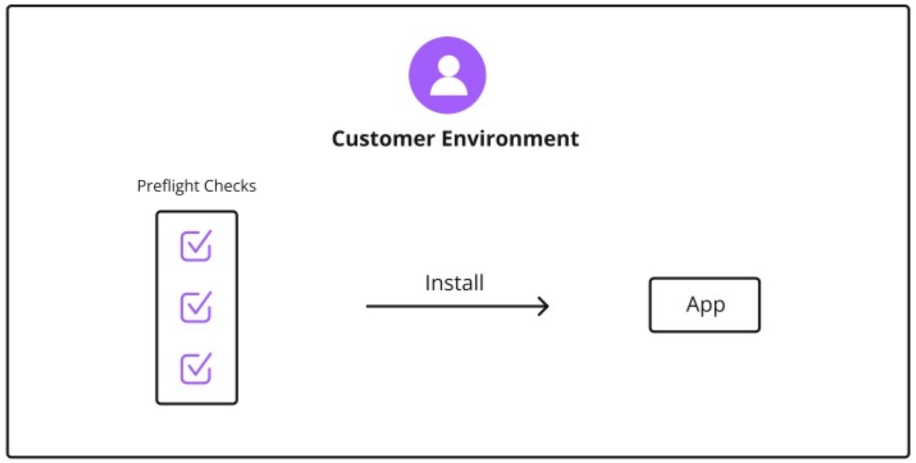
Ultimately, the challenge with installation is that the vendor has to be prepared to meet a spectrum of customer requirements and sophistication. This increases the complexity for vendors who need to consider each installation path in all future releases, testing, updates, support, and so on.

Documentation

To ensure a good installation experience for all customers, vendors should provide detailed installation instructions that explain how each component of the software is configured and installed. This should include information about upgrading or downgrading, proxy installations, advanced configuration options, and any other supported installation path. Additionally, documentation should include any security-sensitive steps (such as credential management, TLS configuration, or network policies) to reduce misconfigurations.

Preflight Checks

In addition to thorough installation instructions, the best vendors also provide preflight checks that customers can run to validate if the resources provided meet the hardware, network, and environment requirements for the software before proceeding with installation. These types of checks can help increase the success rate of installations and upgrades, reducing customer frustration and speeding up the time-to-value for the application.



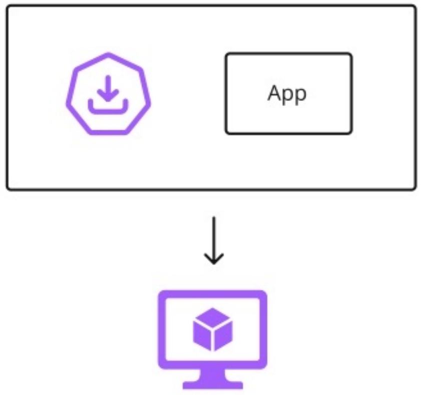
GUI-Based Experience

Providing an installation GUI can also make it easier for less advanced customers to complete installation tasks, such as providing their license or configuring the deployment, without needing to interact with the command line or edit complex YAML files. This can improve the customer experience and cut down on installation errors, helping to reduce the number of support issues related to installation.

Open Standards or Bespoke Installers

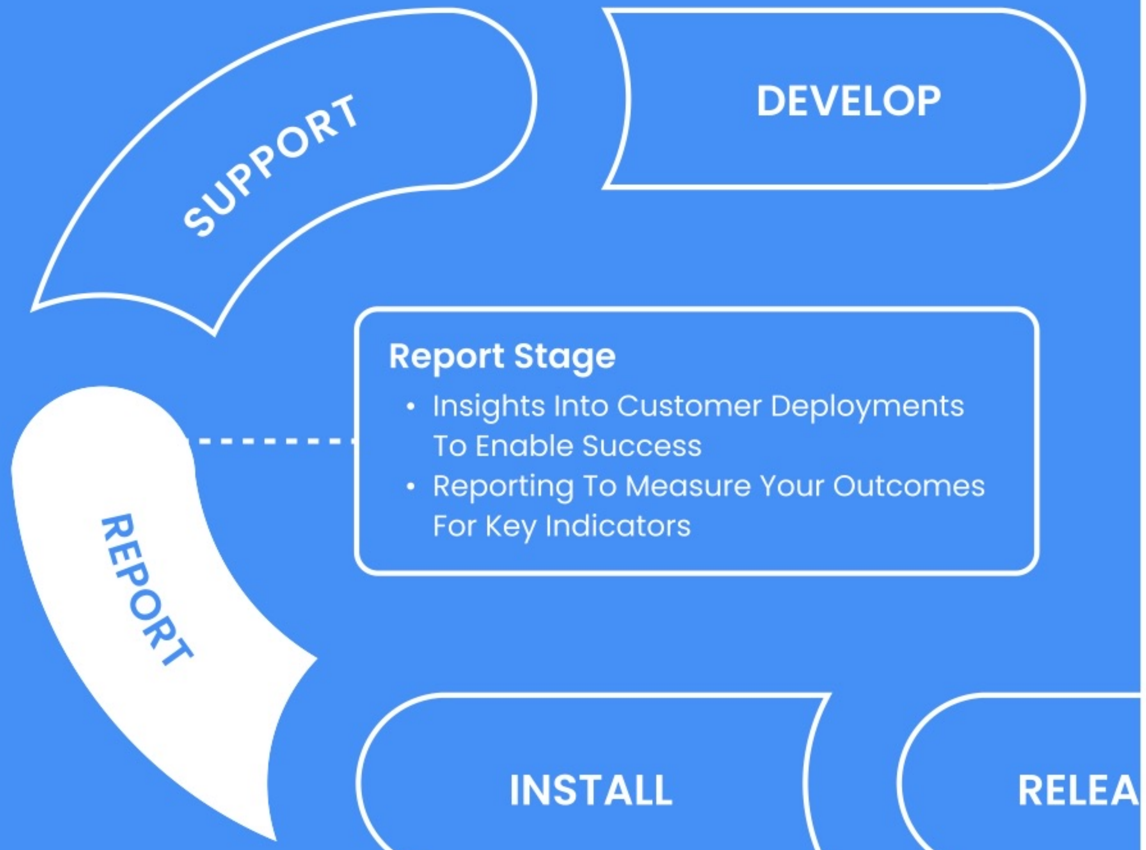
Finally, whenever possible, vendors should utilize existing packaging and installation tooling that is already widely adopted across the industry. For example, Helm is a popular package manager and installer for Kubernetes applications used by many modern enterprises and software vendors. Taking advantage of contemporary industry standards like Helm avoids the overhead of maintaining your own installer, and also ensures that many enterprise customers will already be familiar with the tooling.

That said, many customers might not be proficient enough in contemporary tools (like Kubernetes and Helm) to successfully install. There might also be customers who would prefer to install on a VM or a dedicated Kubernetes cluster rather than attempting to install to an existing shared cluster. To address these use cases, many vendors include a Kubernetes installer that delivers Kubernetes alongside the application so that customers can install on a VM or bare metal server. In this case, vendors must ensure that any installer artifacts are either packaged with the corresponding application release or are published separately where they can be accessed by customers.



Providing a bespoke Kubernetes installer puts the burden of Kubernetes management on the vendor, as their customer will consider it a dependency of the application as opposed to a core system they manage themselves. This also means that it is important for vendors to consider the security posture of the underlying Kubernetes distribution used by their installer. Using a “0-CVE” Kubernetes distribution such as Replicated's Embedded Cluster that minimizes dependencies, prioritizes frequent updates and patches, and implements security best practices can help reduce the overall attack surface for the installer, which could lead to fewer security vulnerabilities.

Commercial Software Distribution Lifecycle



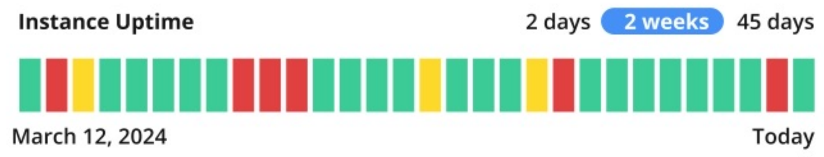
Report Stage

- Insights Into Customer Deployments To Enable Success
- Reporting To Measure Your Outcomes For Key Indicators

Report

Reporting refers to gaining visibility into performance and usage metadata for software instances running in customer-controlled environments. For example, many vendors will collect:

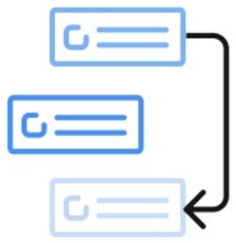
- Metadata about the environment where the application is running, such as the Kubernetes distribution, Kubernetes version, or cloud provider
- Application uptime and service status
- Adoption data such as the current application version
- Usage data such as daily or weekly active users of the application



Software Usage and Adoption

In contrast to traditional observability, which often includes a firehose of logs or key-value pairs from a database, the goal of reporting for modern enterprise software is to provide insight on application usage and functionality. This type of insight-driven reporting is often described with terminology such as telemetry, phone home, or heartbeat.

For vendors, access to reporting data empowers the team to take more informed action:



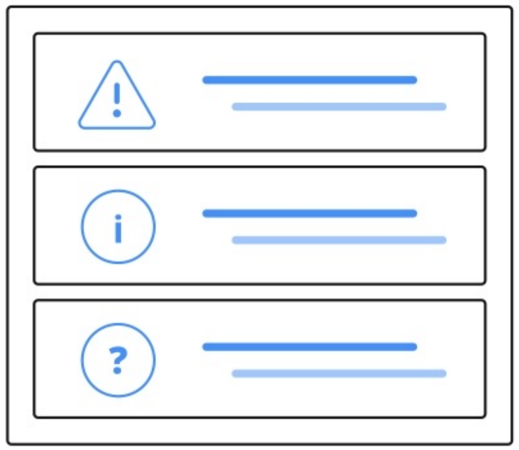
- Usage data can inform prioritization decisions about feature development. For example, low feature usage can indicate the need to invest in usability, discoverability, documentation, or in-product onboarding
- Adoption data, such as the version that each customer running, can be used to understand and monitor the CVE data for each customer's deployed instances
- Decreased or plateaued usage for a customer can indicate a potential churn risk, while increased usage for a customer can indicate the opportunity to invest in growth, co-marketing, and upsell efforts
- Performance data, such as simple uptime data, helps more quickly troubleshoot and resolve issues. Uptime data can also help vendors understand the resiliency of their software, which is important for both operability and security

Enterprise customers also often expect access to this reporting data through formats such as dashboards, data exports, reports, or notifications. For example, customers like to see their usage data to make sure they're within their contractual limits. This is especially useful for air gap customers, who will typically self-report when they are over usage limits in order to get back into compliance with the contract.

Scope and Prioritize Security Issues

Having a robust reporting framework that is associated with each customer's unique license can also be critical for scoping, prioritizing, and communicating security vulnerabilities to enterprise customers.

When there is a known CVE in a third-party library that's part of the software supply chain, or when a security bug is discovered in the application itself, vendors need a way to identify which customers are affected and to what severity. Usage and adoption data such as the version that each customer is running and which features each customer has access to can help vendors determine who they need to notify and how to prioritize fixes across customers. For example, if there is a vulnerability in a specific version of the software, then vendors can notify the affected customers before publishing a public disclosure. Or, if certain customers have features or entitlements that reduce their actual risk, then vendors can tailor the message they send to those customers accordingly.



Transparency, Privacy, and Optionality

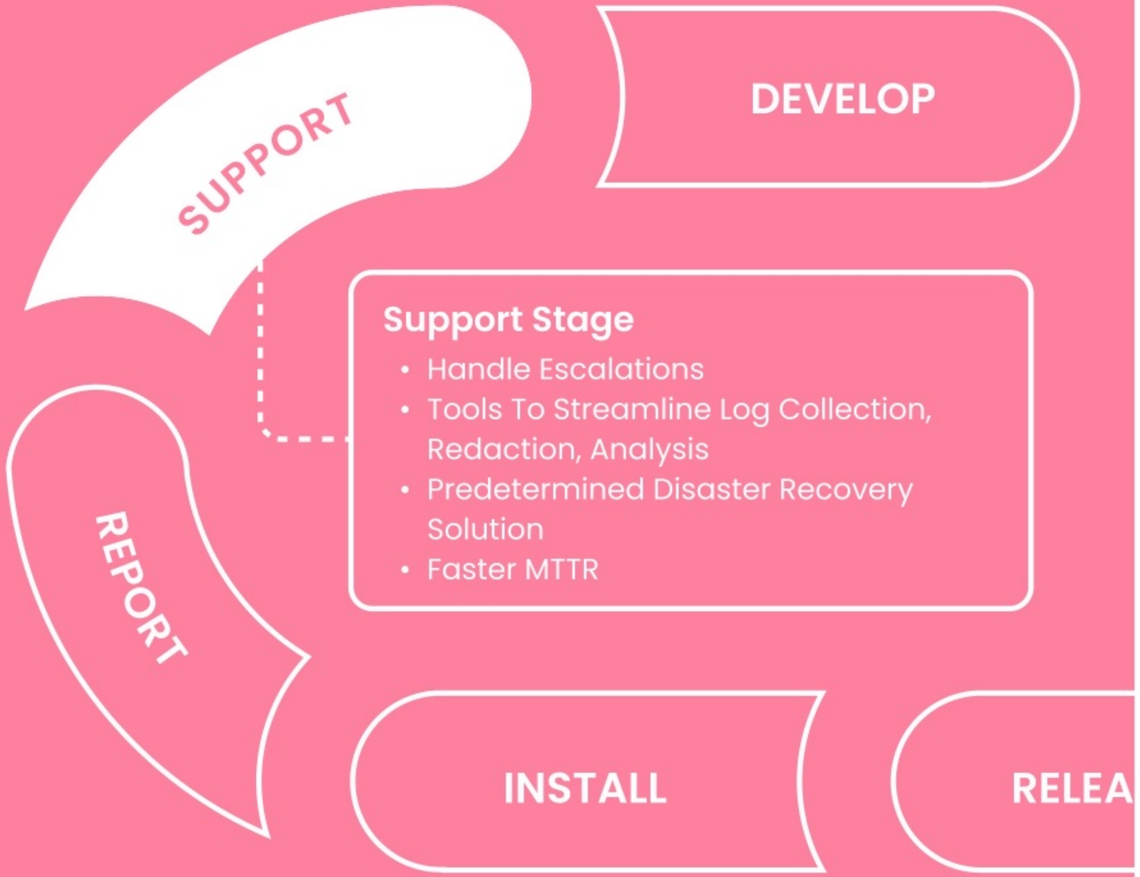
When reporting on instances of enterprise software, vendors should adhere to principles of transparency, privacy, and customer choice:

- Publicly disclose what data elements and fields are transmitted. Not only do enterprise customers appreciate this information up front, but many will require it according to their compliance framework
- Redact sensitive data (such as database connection strings, passwords, or other API tokens) from being sent back to the vendor environment
- Give customers the opportunity to opt in or out of sending different types of data. For example, some customers might opt to send nothing, or to send only diagnostic data used for support purposes

This helps to build trust between the vendor and customer in that security-minded enterprise customers can be assured that the vendor will not collect more data than was agreed to, nor expose them to potential security or privacy concerns.

Transparency, privacy, and customer choice are also critical for reporting on air gap instances, which present unique challenges due to the lack of outbound internet access. Software vendors can collect reporting data for air gap instances by providing opportunities for customers to send redacted data when opening a support request, or through regular surveys. Collecting reporting data from air gap environments on a regular basis (such a monthly or quarterly) can help give software vendors a more complete picture of how customers are using their software, while providing air gap customers valuable insight into their usage.

Commercial Software Distribution Lifecycle



Support Stage

- Handle Escalations
- Tools To Streamline Log Collection, Redaction, Analysis
- Predetermined Disaster Recovery Solution
- Faster MTTR

Support

Support refers to the services, tools, and documentation offered by a software vendor that help customers troubleshoot and resolve issues with their instance of the vendor's software. Providing robust support ensures that issues are resolved quickly, reducing interruptions to usage.

For enterprise software, the scope and breadth of support services provided by the vendor are often defined in a Service Level Agreement (SLA). For self-hosted software, SLAs typically define service agreements for support, such as maximum wait time for addressing issues, maximum response time for support requests, standard support hours, and emergency support. For example, 24/7 support hours and a response time of less than three hours for the most critical support issues are both common expectations for enterprise customers.

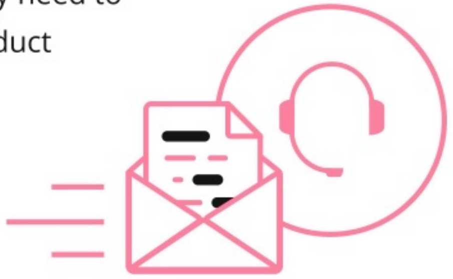
Support teams for modern enterprise software aim to reduce the mean time to resolution (MTTR) for support issues while also meeting the agreements defined in the SLA. To be successful, support teams need:

- Global coverage that enables 24/7 standard support hours
- The necessary training and expertise to address customer issues

- Access to the right diagnostic information from the customer environment, such as support logs, the Kubernetes distribution and version, and usage data

Accessing diagnostic information from the customer environment is challenging for on-prem software because the environments are often disconnected. This means that support engineers cannot simply SSH into machines or view a stream of observability data. Instead, software vendors can provide tools that securely collect redacted information from the customer environment and run diagnostics. This type of tooling can not only be used to generate troubleshooting suggestions for the customer, but can also provide the option for the customer to send the diagnostic information back to the vendor for additional support. Similar to collecting reporting data from customer environments, it is important that any such support tools redact sensitive data, and that vendors are transparent about the types of data collected.

Apart from support tools and services, high-quality documentation and community-based help articles are also critical for providing customers with the information they need to self-resolve issues. Keeping the product documentation and help articles up-to-date with troubleshooting information for common support issues helps to avoid multiple different customers running into the same issue, saving time and frustration.



Conclusion

Distributing software into self-hosted environments poses unique challenges that require unique solutions and guidelines. By following the steps outlined in the Commercial Software Distribution Lifecycle, software vendors can ensure that enterprise customers get the best possible experience, product teams lead the way towards the most impactful new features, and development teams prioritize fixing the right problems.

Develop with the enterprise customer in mind.

Test across as many unique environments as possible.

License software with simplicity and customization.

Release the right software to the right people using a consistent approach.

Install by meeting the customer where they are.

Report to surface valuable insights to internal teams and customers.

Support complex problems with clarity and speed.

Create world-class self-hosted software.

Assessment

Want to see where your team stands in developing self-hosted software against the Commercial Software Distribution Lifecycle?

Go through this assessment together to analyze your software maturity.

Develop

0	App Contains Thousands Of Services, All Manually Deployed, No Automation
1	App Is Portable
2	App Is Portable And Customers Can Swap In Databases, Statefulsets, Infra Components They Manage
3	App Is Resilient (Comes Back On Failure)
4	App Is Highly Available (Avoids Failure)

Test

0	Minimal Testing Outside Of Unit And Functional Tests Of The Application
1	Test Installation Into Customer Distros And Versions
2	Add CNI, CSI, CRI (Addons) Into Matrix
3	Add Upgrades For Customer Tests
4	Add Customer Representative Data Into Test Matrix

License

0	No Licensing
1	Unique Credentials For Registry
2	License Key That The App Validates To Run
3	Signed Values For Entitlements, Set Expiration Dates, Update
4	Full License Management System With Advanced RBAC For Images, Easy To Update Entitlements And Sync To Customer Instances

Release

0	Release A Version, Customers Find It
1	Notify Customers On Release
2	Assign Customers To Channels, Work With Various Release Cadences
3	Required And Skippable Versions
4	Support For Collecting Telemetry Data From Airgap Installations

Install

0	Customers Don't Have Choice (Need Specific K8s, Or OS)
1	Customers Can Bring Any K8s, Installs With Helm
2	Helm Plus An Embedded Cluster Option
3	Add Airgap Installation Support
4	Support For Existing Environments That Aren't K8s (Docker, Nomad, ECS)

Report

0	No Visibility Into Customer Environments
1	Basic Telemetry Collected For Online Installations (Installed Or Not, Version, Ip Address, Cloud Provider)
2	Specific Operations Telemetry (Pod Status, Restarting, Deployment Rollout Status, Etc)
3	Add In Custom Metrics & Telemetry For Usage Reporting
4	Support For Collecting Telemetry Data From Airgap Installations

Support

0	Customers Grab Logs And Send Them Upon Request
1	Predefined Script To Generate Common Logs
2	Log Gather Tool That Provides Redaction Capabilities
3	Log Gather Tool That Redacts And Analyzes To Give Customers Insight
4	Regular Snapshots Stored And Team Of 24/7 Experts Available To Solve Complex Environmental Issues

Take the number you selected from each section and add them together. Where your number falls in the below ranges indicates the quality of your software distribution architecture and how you can use this information to improve or exemplify your process.

0-7	There's a lot to be desired in your software distribution architecture. Make sure to spend time reading through each section of this book to learn how to improve.
8-16	You have some of the basic building blocks, but there's still a lot of work to do to ensure you're providing a successful installation experience for all your customers.
17-23	You're on the right path, and have put a lot of great work into ensuring that you can distribute to many different types of customers. Still, you probably hit customer installation blockers relatively often and your self-hosted deployments could be improved.
24-28	You have set up an amazing software distribution architecture, and you're most likely successfully handling many different installation methods for your customers.

Great job! You've completed the assessment!

Want help improving how you distribute your application to self-hosted customers? Check out Replicated.com

About the Authors

Grant Miller

Grant Miller is the CEO and Co-Founder of Replicated. He has 20+ years of experience in the technology industry, and is also the creator of EnterpriseReady.io. When not creating products, podcasts, and articles about enterprise software, he's traveling and exploring with his wife and two kids. He currently lives in Austin, TX.

Paige Calvert

Paige Calvert is the Documentation Manager at Replicated. She has 10+ years of experience as a technical writer in the enterprise software space. Outside of tech comm, she enjoys playing volleyball, testing out new recipes, and watching live theater. She currently lives in Denver, CO.

Kaylee McHugh

Kaylee McHugh is the Director of Marketing at Replicated. She has 10+ years of experience in the technology industry with roles ranging from software engineer to CEO and everything in between. In her spare time she enjoys hiking, snowshoeing, and cozying up with a good book. She currently lives in Seattle, WA.

REPLICATED

replicated.com