Version control for Salesforce

A practical guide to implementing git-based release management

Release management made easy | **gearset.com**

Contents

Introduction 2
Who is this whitepaper for? 2
A brief introduction to version control 2
Definitions 3
The path to version control 4
- Production development 4
- Sandbox development 5
- Version control development 6

The benefits of version control in Salesforce development 6
Why don't more Salesforce teams use version control? 8
- Limitations of the first-party tooling 8
- Thinking version control is only for enterprises 8
- Not knowing what good looks like 8
- Technical barriers of command-line tools 9

Getting started with version control: introducing Git 9
- Service providers 9
- On-premise vs hosted 9

A best practice development model for Salesforce 10
- Overview 10
- The development model 11
- Branch management 12

Dealing with hotfixes 12
- The hotfix model 13

What metadata to version control 14
- Start with a controlled subset 14
- Managed packages 15

Repository configuration 15
Finding the right deployment tool 16
- For developers 16
- For admins and release managers 16
- For team leads and architects 17

Conclusion 17
About Gearset 17
Further reading 18

Introduction

Version control is one of the most powerful tools development teams can leverage on their path to effective release management, yet its adoption in the Salesforce ecosystem is surprisingly low. In this whitepaper we'll examine how version control works, the benefits of version control over in-org development, and introduce a best-practice model for implementing version control in your business.

Who is this whitepaper for?

Anyone involved with the administration, development, maintenance, or management of Salesforce environments, looking for ways to improve the cadence, simplicity, reliability, and auditability of their release management.

From the fundamentals of version control through to a detailed release management model for teams of all sizes, this whitepaper contains best-practice advice for developers, administrators, team managers, and technical architects alike.

A brief introduction to version control

Version control is a well-established concept in most software languages and platforms, and is one of the key enablers for efficient Agile development. At its heart, version control focuses on one core concept: tracking changes to files over time.

Version control becomes an enabler for development teams to work faster and smarter, by:

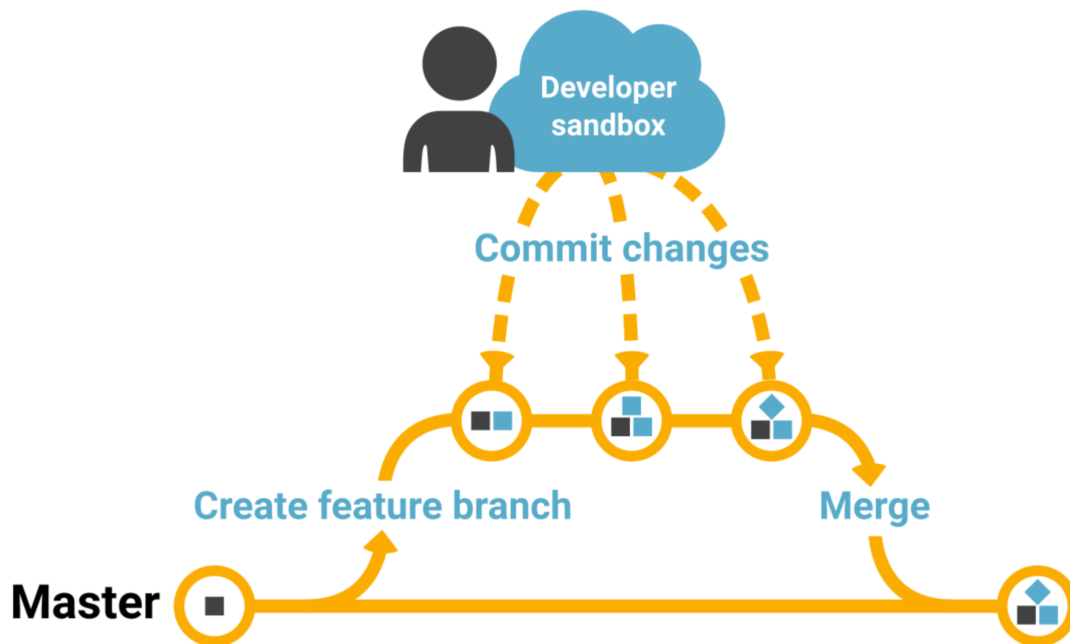
- Being a single 'source of truth' for the team.
- Enabling parallel development streams by allowing developers to work on changes in isolation without the environment changing underneath them.
- Providing tools to identify and resolve file conflicts.
- Maintaining a full audit trail for every stage of the development cycle.
- Facilitating collaboration and code review.
- Making it easy to maintain and deploy different versions of code across test, staging and production environments.

Salesforce teams who use version control release higher quality code more frequently, introduce fewer bugs, maintain better reporting and visibility, and have a better ability to rollback changes. This translates to a better relationship with their end users, and faster project delivery. As the Salesforce modules on Application Lifecycle Management put it: "*Using version control is considered to be a general best practice for software development…and ensures a quality development process*."

Definitions

Before we go any further, let's define a few terms we're going to use throughout this whitepaper.

- **Version control / source control** systems provide a mechanism for tracking changes to files. For software projects such as Salesforce, this almost always means representing configuration changes as text files (Apex / XML) and tracking changes to those text files over time. The terms version control and source control are often used interchangeably.
- **Repositories** are the containers version control systems use to store files and track changes against them. Repositories also provide a mechanism for teams to share changes, review each other's work, and resolve conflicts. Repositories are based around a central *master* store of files, with a number of branches containing new features in development.
- **Branches** provide independent working environments for developers where new features can be built and tested in isolation from other development work. Branches are part of the everyday development process with version control. Branches allow developers to work in parallel, review other's work, and control when changes are released for testing.
- **Commits** are collections of changes which a developer adds to a branch to record their development progress. As a developer works, the version control system automatically tracks changes they make to files. When a portion of the feature is complete, the developer submits a collection of changes, accompanied by an informative description, as a commit to their branch.
- **Merging** is the process of integrating changes from one branch to another, including into master. Merging is typically done when a feature is ready for user testing in Salesforce orgs, and usually involves code review by other members of the development team.
- **Pull requests / merge requests** are initiated when a developer wants to merge a branch. They provide a quality and compliance gating process before new changes are integrated into the main code base or released to Salesforce environments. All changes can be easily compared, and code reviewed and commented on by peers. The final approval and merge is completed by someone other than the developer who worked on the changes. The terminology for this process varies from version control provider to version control provider, but the concept is the same.



An overview of development using version control

The path to version control

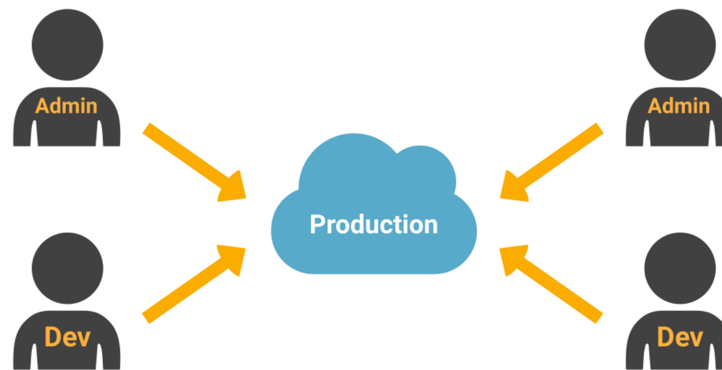
Salesforce powers thousands of businesses around the world, from small non-profits to global technology giants. Despite this diverse customer base, we have observed that many businesses follow a similar evolution in their release management, motivated by the need to effectively manage the increasing complexity in both their teams and environments. This evolution falls into three broad categories: production development, sandbox development, and version control development.

Production development

Initially, changes are made directly in the production environment. This can be a quick and efficient way of working after the initial implementation of Salesforce if the business doesn't have sufficient customization of their organization to require a development team, or if costs need to be kept to a minimum, as there's no need for additional sandboxes.

While simple, this is a risky approach. Changes are being made to the live organization that's being used by the business on a daily basis. Bugs and unfinished features can cause significant disruption, and it's difficult to quickly identify and fix issues. Parallel development streams are very challenging, with developers or admins commonly overwriting each other's changes. There are also fundamental limitations in terms of the changes that can be made – new Apex classes can't be created in a production environment, for example.

Production development flow



A typical production development flow

Sandbox development

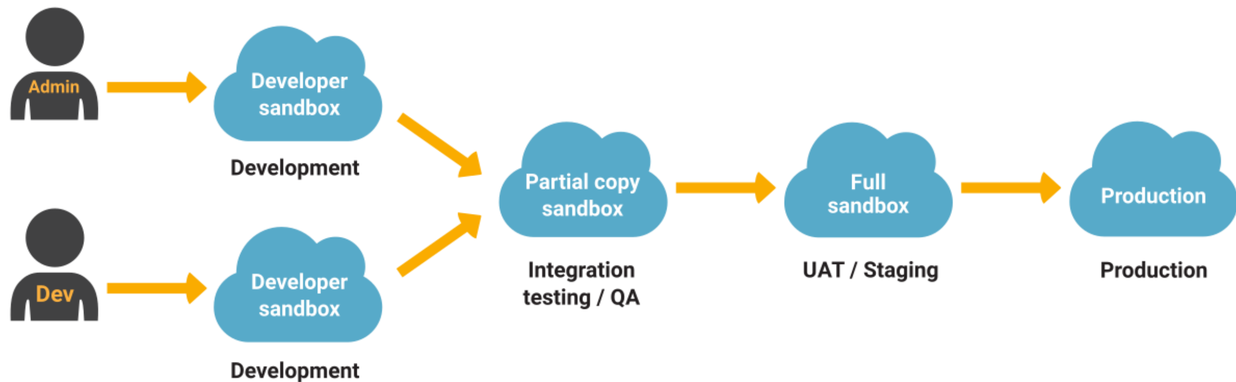
To avoid the issues with working in production, teams commonly move to using a number of sandboxes to build and test code prior to release into production. This can vary from a single sandbox to a range of developer, partial and full sandboxes with increasing complexity and data.

Development in sandboxes provides many advantages over development in production. Developers and admins can begin to work in parallel development streams, reducing the chances of stepping on each other's toes. Changes can be tested prior to release to production, reducing the risk of bugs and making it more likely a feature will meet the business needs of the end user. Apex code can be written and tested before being migrated to production, allowing for a much greater degree of customisation of the environments. The segregation of different environments also allows for more refined change control. As a result of these benefits, many teams quickly move to the sandbox development model, adapting the process as the team size and environment complexity grows.

Sandbox development does not solve all the issues of a production-centric approach, however. Code conflicts and accidentally overwriting changes remain a pain point, and there is limited audit capability to track why and when changes were made.

It also introduces several new challenges which teams must contend with. With parallel development streams across multiple environments it's common for sandboxes to become out of sync. This limits the effectiveness of testing, and can increase the risk of duplicated work. Identifying which features are ready for testing, and which components should be included in each release often becomes a manual, spreadsheet-based task. In the event of a rollback, it can be difficult to know exactly what the most recent stable state was. The biggest challenge for many, however, is simply migrating the metadata effectively between the multiple

environments. The complexity of Salesforce metadata, combined with the steep learning curve of the command-line based deployment tools can become a serious time drain on the release team.



A typical sandbox development flow

Version control development

The benefits of sandbox development and the ease of adoption means most teams have moved onto this approach. Despite its advantages, its inherent challenges limit how far teams can go on their path to Agile development. Over time, the lack of automation, limited change tracking, interfering development streams, and complex environment management drive many teams to look for a better solution. For many, version control is at the heart of such a process.

The benefits of version control in Salesforce development

Version control has a range of benefits over in-org development which makes it a great choice for small and large development teams alike.

Single source of truth

The master branch of a repository becomes the source of truth for tracking changes and resolving conflicts. Everyone in the development team can see at a glance what code is currently ready for release and what is being worked on in branches. This simplifies releases and ensures the whole team is aware of what is currently live.

Manage code conflicts and deployment risk

Version control systems bring tooling that can help teams identify and manage conflicts early - during merging, rather than at deployment time. This means by the time they deploy, they're

already confident that they're pushing exactly what they want. The importance of this can't be overestimated - with in-org development, finding and resolving conflicts would only happen at the point of release to sandbox or production. Release windows are often tight, and unexpected delays can have big impacts on the business. Using version control, teams can handle conflicts when merging branches, and the result is ready to be deployed completely as-is to any subsequent environments for testing. This happens as a natural part of the development workflow, rather than a week or two down the line when it's being deployed.

Enable parallel development streams

By allowing each developer to work in their own branch, multiple development streams can easily be maintained. Teams no longer have to worry about another user accidentally modifying an item they are working on, or trying to disentangle their changes in a shared sandbox. This also helps separate feature work from hotfixes.

Maintain a full audit trail

Through commits, all changes are annotated and associated with a team member so the whole team knows who did what and why. This can be invaluable if they ever need to go back to review historical work or bring new team members up to speed, and is a requirement of some compliance regulations.

Reduce bugs through code review

During the merge process, all changes should have an associated assignee so they can be reviewed before being pushed into production. This peer review of code enables teams to find and fix bugs prior to release and increases overall code quality.

Deploy consistently across environments

Branches provide stable release points to quickly and reliably propagate the same set of changes to staging, UAT, and finally production environments. Teams can rely on the knowledge that they're thoroughly tested exactly what will be released, without the need for manually updated deployment tracking documents.

Simplify rollback

In the event of a rollback, it's simple to revert a merged branch and return an org to its previous state. Through the detailed history of changes, and the merging process, teams can then go back in time to understand why things went wrong, and identify who knows most about each part of the system to resolve the issues.

Release faster

Version control encourages an increased release cadence, deploying smaller number of changes more frequently than is traditionally possible with in-org development. With a well-

designed process which doesn’t slow teams down, this reduces risk while enhancing the speed of project delivery.

Why don’t more Salesforce teams use version control?

With all the benefits of version control over in-org development, one might expect the majority of Salesforce development teams to have adopted this approach. But the numbers are surprisingly low, with most using the sandbox model. There are four main reasons for this.

Limitations of the first-party tooling

Salesforce has a unique development process. With the more traditional platforms, the code is the specification of an application, and the application is built straight from that blueprint. When changes have been made, developers take the updated blueprint, compile the application from it, and deploy it.

In contrast, with Salesforce, the application is a living, changing thing in its own right - "clicks not code" means the application can exist entirely without a user-facing source code (metadata) representation. Rather than compiling the application from source, the source is generated from the current state of the application. An updated version of the application isn't created and deployed, it's modified in-place either directly, or via an API.

Because of this ability to modify the Salesforce org in-place, there hasn’t been the same urgency for tools to handle Salesforce source code. Consequently, the first party tools’ support for extracting metadata and deploying changes are very limited, and this has hindered wider adoption of version control.

In the past few years, more advanced deployment tools have been created to fill this gap. These third-party tools enable Salesforce developers to enjoy the same level of metadata migration and management that is common on other platforms, while solving the nuances of working on Salesforce.

Thinking version control is only for enterprises

Many teams consider version control a solution that’s only appropriate for large enterprises, and not something a small development team can take advantage of. As we will set out below, with a little planning, version control can bring a range of benefits to teams of all sizes.

Not knowing what good looks like

Due to the relatively small numbers of Salesforce teams using version control, there is often a lack of understanding of what a *good* process should look like. The most common pitfall is over-engineering a process which often ends up becoming more of a hindrance than a help. Teams bogged down in an overly complex process often revert to their old ways of working.

Technical barriers of command-line tools

For admins or developers who aren't familiar with version control or command-line tooling, there has traditionally been a lot to take in. Learning how to use the command line and edit code by hand so teams can commit changes to version control is a hard sell when they know how to make a change via the Salesforce UI.

Luckily, release management with version control doesn't have to be difficult. With a good release model and a modern deployment tool, it's easy for teams to create a simple yet effective process which works for developers, admins and managers of all experience levels.

Getting started with version control: introducing Git

The first step to implementing version control is choosing which system to use - a few common types are Perforce, Subversion (SVN), Mercurial and Git. Deciding which system has a big impact on the process and tools available to the team.

Luckily, it's a simple choice for Salesforce development. In recent years, Git has established itself as the *de facto* version control system. Git is used by millions of developers around the world and is the recommended system for all Salesforce development teams.

Service providers

Although there are many services providers for Git-based version control, there are three clear market-leaders:

- GitHub [https://github.com/]
- Bitbucket [https://bitbucket.org/]
- GitLab [https://about.gitlab.com/]

All three provide a mature and effective platform for Salesforce development. Which provider to use will come down to the specific needs of the team and personal preference.

On-premise vs hosted

Most version control systems have the option of running either as a hosted service, where the provider manages the hosting and provision of the software for you, or running locally on your own systems, known as on-premise.

Hosted version control provides the greatest flexibility, stability, and ease of setup. On-premise has the advantage of more discrete data control, but that is offset against the additional cost, configuration, and infrastructure investment required to set up and maintain it.

For the vast majority of Salesforce teams, the hosted version of a version control system is the best choice. The established Git providers are trusted by some of the largest companies in the

world and have extremely robust data protection and uptime policies. If opting for an on-premise solution, it is worth considering the additional costs and how access to hosted systems, such as Salesforce or Gearset, will be managed through corporate firewalls.

A best practice development model for Salesforce

With a Git-based version control system in place, it's time to introduce our development model for Salesforce teams using version control.

Overview

Our model has been designed with three key guiding principles in mind:

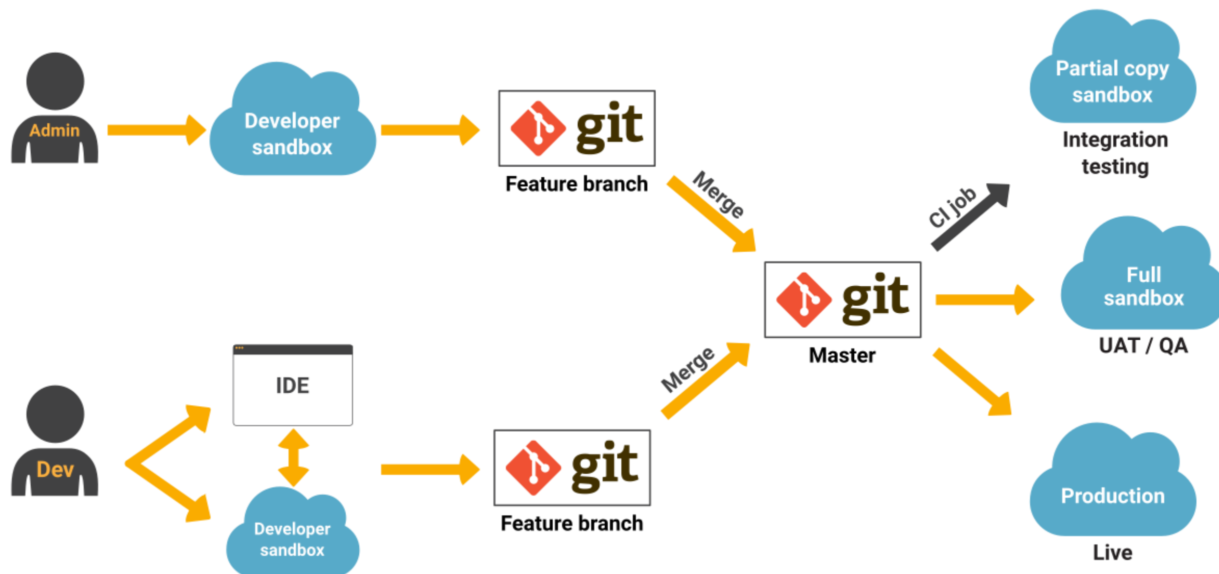
- **Simplicity**. Complicated processes drive people crazy. Keeping things simple not only improves development speed, it also makes maintenance and training much easier.
- **Practicality**. Our model is tried and tested through years of development experience, yet flexible enough to be tailored to the needs of different businesses.
- **Designed for Salesforce**. The model is designed to cater for the nuances of development on the Salesforce platform.

At a more practical level, there are a number of guidelines to consider when implementing the model:

- New features and bug fixes follow the same development flow.
- Master should always be deployable, and is always treated as the single source of truth.
- New changes should be deployed out to your testing environments as soon as possible after merging.
- Branches should be used to represent a single deliverable request from the business, such as a new feature, user story, or bug fix. Include the minimum viable number of changes in any one branch and nothing more.
- Frequent, small releases are better than infrequent, complex ones. The longer a branch exists without getting merged, the greater risk for merge conflicts and deployment challenges.
- Never automate deployments to production - human overview is always advised.
- Keep things lean to remove barriers to adoption. A process is no good if no-one adheres to it.

The development model

This model is a framework for how we think a best-practice release flow should integrate with version control. Of course, each business is different, and the model is designed to provide flexibility for teams to customise to suit their specific regulatory or business needs.



Step 1 - Environment setup

- Create a new developer sandbox or refresh an existing one from your master branch or production environment. This gives all developers a consistent starting point.
- Create a new branch from master on the developer's local machine. Name it descriptively, either referencing your user story or the bug being fixed (e.g. `feature/user-story` or `bugfix/fix-account-visibility-for-sales-profile`).
- Publish the branch to your central repository.

Step 2 - Development

- Developer(s) work on the changes in their developer sandboxes, or using a local IDE and a sandbox. Tests should be run locally on a regular basis to check for regressions.
- Changes are periodically committed to their branch and published to the central repository. Publishing the branch gives visibility to other developers and provides backup in the event of local data loss.
- When the changes are complete, a pull request is opened to merge the branch back into master. All changes must be reviewed by at least one assignee prior to merging. This helps maintain code quality and catch bugs. The notes and feedback during a pull request also produce an audit trail if the changes need to be revisited in the future.
- After review, the branch is merged into master.

Step 3 - Release

- A CI job detects the new changes in master, and automatically deploys them out to an integration testing sandbox (usually a partial copy) for rapid testing.
- Any test failures or changes via user feedback should be fixed, committed, and submitted via a pull request.
- After passing integration testing, deploy the changes from master to UAT / QA environments in turn. The number of environments between integration testing and production will vary based on company and team size, but your final pre-production environment should at the very least be a partial copy sandbox which is in sync with production.
- After final approval, the changes are pushed from master to production. Releases to production should always be a manual process to ensure appropriate oversight.
- Deployment reports from the final release should be stored with any user stories to add an extra level of audit trail for business owners.

Branch management

There are two approaches to managing branches created during development.

- **Branches are deleted from the central repository as soon as they are merged into master**. This keeps your repository simple and easy to maintain. If a bug or change is discovered during testing, the branch can either be reverted through your version control system, or new commits pushed from the local developer environments. This works well for smaller changes which are released more frequently.
- **Branches are deleted upon deployment to production**. This makes it easier to make changes to a feature while it goes through testing, but care should be taken to delete the branch after final release to avoid redundant branches cluttering up the repository. This approach works well for larger, more complex releases which may be on a slower release cadence.

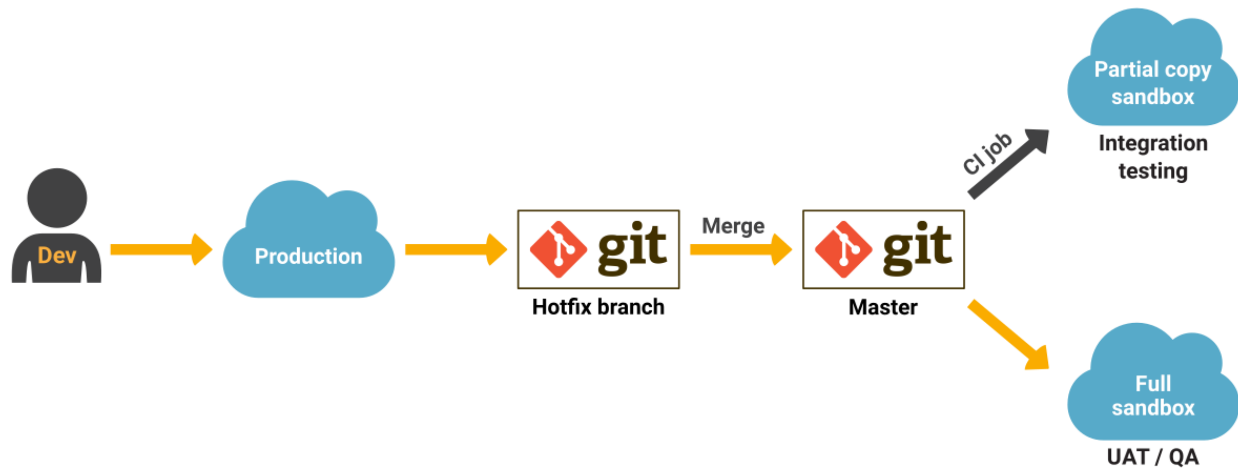
Dealing with hotfixes

No matter how good the release model, changes will sometimes be made outside of the defined process. An urgent bug fix in the production org may need to be rushed through to release, or an admin may make a small configuration change in a UAT org through the Salesforce UI.

The ability for users to rapidly make these small changes is one of the strengths of Salesforce and allows businesses to respond to user requests more quickly than is typically possible with other platforms. Rather than attempting to stop users working this way, a good release process should have an effective way of quickly identifying these hotfixes and incorporating them back into the defined development model. Tracking hotfixes in this way avoids them being overwritten during the next scheduled feature release.

The hotfix model

This lightweight process is designed to rapidly incorporate any changes made outside of the standard release flow back into version control.



Step 1 - Capture changes

- Create a new branch from master. The branch should be named something descriptive to make identification of the hotfix easy (e.g. `hotfix/prod-account-inaccessible-to-sales-profile`).
- Pull the changes from the org into the new hotfix branch.
- Open a pull request to merge the branch back into master. Teams should ensure that all changes are reviewed by at least one assignee prior to merging. The notes and feedback during a pull request also produce an audit trail if the changes need to be revisited in the future - especially useful for hotfixes which are generally undocumented and may not have an accompanying user story.
- After merging, the hotfix branch can be deleted.
- A CI job will now automatically push the changes into your integration sandbox.

Step 2 - Propagate changes

- After testing in integration, manually deploy the changes to your other environments. As any non-developer environments should be based off master, there should be no additional conflicts beyond those identified in the steps above.
- Notify developers working in feature branches of the change so they can assess any impact on their current work. Usually, any conflicts will be resolved at the pull request phase, but developers may wish to rebase their branches if the changes directly affect their ongoing work.

It's well worth setting up an automated way of tracking changes to your primary Salesforce environments (UAT/QA/Prod) to allow teams to proactively manage this process. By comparing org snapshots on a daily basis, teams can rapidly identify and incorporate hotfixes as they are made.

What metadata to version control

When starting out with a version control, it can be tempting for teams to immediately put all of their metadata into the repository as a form of production backup.

There are a few problems with this approach:

- While the majority of metadata can successfully be managed in version control, some types, such as *Site.com*, do not lend themselves to version control due to automated changes made by Salesforce. These will always be out of sync with the repository.
- Some metadata can be undeployable once it has been removed from an org, due to Salesforce API limitations.
- Continuous integration jobs are much easier to manage with a subset of metadata.
- The high volume of metadata can be overwhelming to begin with, and the burden of its management can slow down the development cycle.
- Typical development flows will not affect every metadata type, so there is little point in version controlling them.

Version control is designed to help teams create, track, and deploy new features. It should not be seen as a backup of your Salesforce environments for disaster recovery purposes.

Start with a controlled subset

Small, regular deployments and rapid testing of new features through continuous integration are core parts of the model. The key to enabling this approach in Salesforce is starting with a limited subset of metadata which can be deployed with a high degree of reliability. This will allow the team to build confidence in the process during its adoption.

A common mistake when designing a release process is to add too many components at once. Each time a deployment challenge is encountered, it erodes confidence in the process and reduces the desire to pursue this approach. Starting with the most important subset of metadata means teams will start seeing success immediately, and any challenges faced when expanding the process to incorporate more metadata types will be more manageable.

Ultimately the process needs to work for the team - if it's a huge effort to set up before any benefits are realised, there's a risk that the process will fall by the wayside. If a team can start seeing even small benefits quickly, then it'll gather steam of its own accord.

An example of the metadata types a team might start adding to their version control and continuous integration jobs might be:

- Apex class
- Apex component
- Apex page
- Apex trigger
- Custom object
- Global value set
- Profile
- Standard value set

Once the team has a reliable end-to-end release flow with this set, they can begin adding additional metadata types as required.

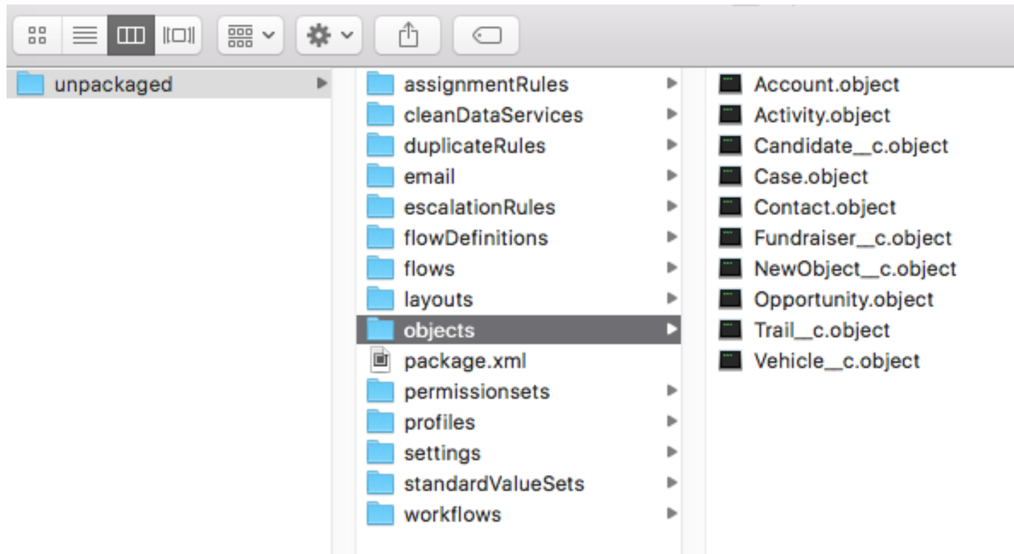
Managed packages

Managed packages behave differently to most other metadata. As a rule, there are two approaches to dealing with managed packages in version control:

- If the packages are not being modified beyond the original install, only the *Installed package* type needs to be added to version control. This is because the installed package effectively provides everything an org needs to ensure that managed packages are in sync between source and target.
- If modifications are being made to the package, both the *Installed package* and the modified metadata must be added to version control to allow the team to track changes. Given the number of changes associated with managed packages, it is advisable to treat them as distinct features, tracked in their own branches. This will help teams spot changes being made through package upgrades versus other feature development work.

Repository configuration

Metadata stored in version control should be structured in the standard Salesforce format, as used with the Ant migration tool. Within each branch (or master), each metadata type sits within its own folder. All of these folders can be placed in a parent *unpackaged* folder, but there should be no further nesting than that. If a *package.xml* file is being used, it should be stored in the top level of the branch, alongside the metadata type folders, as per the screenshot below.



How to structure metadata in your repository

Finding the right deployment tool

The simplicity and effectiveness of the model is predicated on being able to quickly move changes between Salesforce and version control, including via automated continuous integration jobs. Picking the right deployment tool, with a balance of functionality and ease of use, is crucial to realizing the key benefits of version-controlled deployments. A great tool will have features which enable all members of the team to work more effectively.

For developers

- Compare repositories and orgs to see the line-level XML differences.
- Granular deployment control to manage exactly what is committed to source control.
- Ability to save and share changes with other developers prior to release.
- Automatic package creation and dependency analysis, including for destructive changes, to remove manual steps and speed up testing and deployments.
- Commit changes to branches directly through the tool.
- Ability to trigger CI jobs off commits to version control.
- Automated org test execution / regression testing to proactively find and fix test failures across all environments before they become blockers at deployment time.
- Unlimited connections to orgs and no packages to install for easy setup and maintenance across multiple development environments.

For admins and release managers

- Quickly visualize what’s different between orgs with change highlighting.
- Commit changes to version control from within the tool, without having to learn the intricacies of the version control system or command line.

- Metadata dependency analysis to simplify and speed up the creation of valid deployments, with ability to work around common deployment blockers.
- Automatic change tracking and alerting for multiple orgs to track hotfixes proactively.
- Shared history and status of CI jobs and deployments across the team.
- Deployment rollback in case of accidental changes to orgs.

For team leads and architects

- Dashboard view of the status of all Salesforce environments and tests.
- Detailed reporting and full audit trail of activity for every deployment.
- Effective team collaboration which enhances workflows without getting in the way.
- Easy to master, with little or no training required.
- Usable by all team members, regardless of experience in Salesforce or version control.

Conclusion

Version-controlled development is a powerful tool for Salesforce release teams looking to improve their release management. Through its centralised change tracking, conflict resolution, and peer review, version control enhances the quality and speed with which new features can be developed and released to end users when compared to traditional sandbox development. Limitations of the first party tooling and a general lack of familiarity with good processes have historically limited the adoption of version control by Salesforce teams, but in recent years, third party tools such as Gearset have filled that gap and opened up version control as a viable option for development teams of all sizes. In this whitepaper we demonstrated a best-practice model for how Salesforce teams can implement version control, providing a standardised process for developers and admins to make, track, deploy, and report on changes and new features across all their Salesforce environments. An effective release management tool is a fundamental part of implementing a successful version control process, and we explored the key features teams should look out for to empower all members to benefit from the new process.

About Gearset

This whitepaper was written by the team behind Gearset, the release management tool for Salesforce. The Gearset team have decades of experience in development and deployment across multiple platforms, including SQL Server, Oracle, .NET, Azure, and Salesforce. We helped build some of the leading deployment tools used by 91% of the Fortune 100 to run their release management and achieve world-class continuous delivery. We're a trusted Salesforce partner and our sole mission is to make development on Salesforce ingeniously simple. We built Gearset to solve the challenges Salesforce teams face with release management.

If you're looking for a tool which perfectly complements your new version controlled release process, you can find more information on Gearset and start a free 30-day trial at https://gearset.com or contact us at team@gearset.com.

Further reading

- **Trailhead** - Application Lifecycle management module
  https://trailhead.salesforce.com/en/modules/alm_deployment/units/alm_source_control
- **Salesforce Development Lifecycle Guide** - Chapter 7
  https://resources.docs.salesforce.com/sfdc/pdf/salesforce_development_lifecycle.pdf
- **Salesforce Apex Developer Guide**
  https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_dev_guide.htm
- **Gearset whitepaper on release management** - Simplifying Salesforce release management: a best-practice approach
  https://gearset.com/assets/whitepaper-simplifying-salesforce-release-management.pdf

Release management made easy | **gearset.com**