Whitepaper on Release Management & Branching Strategy

By: Jatin Vyas

Document Information	
Document Name	Generic Release Management Process
Owner	Jatin Vyas

Document History			
Version	**Date**	**Author**	**Comments**
1.0	07-07-2020	Jatin Vyas	Initial Draft
1.1	11-10-2020	Jatin Vyas	Update the branching strategies
1.2	10-03-2022	Jatin Vyas	Update the process
1.3	15-06-2023	Jatin Vyas	Update the process

Document Review		
Date	**By**	**Comments**
10-07-2020	Sushil Bhosale	Initial Review
05-11-2020	Sushil Bhosale	Second Review and Approved
01-01-2022	DevOps CoE Forum	Initial Review
14-05-2022	DevOps CoE Forum	Second Review
22-07-2023	DevOps CoE Forum	Third Review and Approved

Classification	**Description**
Public	Information is approved for public communication.
Company Confidential	Information that can be circulated at its owner's discretion for employees and authorized third parties.

Management Summary

This document describes a generic Release management and branching process that be readily implemented as a starting point and then be customized to suit specific client processes

The objective of this document is to document a fixed process that developers and release managers use to ensure standardized methods and procedures are used for all Releases.

This document describes, from start to finish, the process to be followed to implement a Salesforce release in any Org.

Introduction

This document is to show release and rollback potential using any CI /CD Tool for Salesforce org and will form part of the Release Management document.

What is Release Management?

The process for release management is to schedule and maintain the integrity of new deployments, all the way from planning to release. In DevOps, the release manager will receive code from the software developers and decide when and how to deploy the release while maintaining uptime for existing services. Both developers and Release Managers collaborate from the beginning of the process to the end – allowing for fewer, shorter feedback loops and faster releases.

Purpose and Benefits of the Release Management Process

Streamlined CI/CD and QA

By moving QA, automation, and testing earlier in the development lifecycle, the salesforce DevOps team can identify potential issues faster. This will reduce the amount of time spent in feedback loops and allow the delivery pipeline to continue moving forward. The more we integrate testing with development workflows, the easier it will be for the team to maintain a consistent CI/CD pipeline.

Reduces downtime and customer impact

By keeping the staging and UAT environments nearly identical to production, the salesforce team will find it easy to identify issues in staging and UAT before deploying the code to production

Use Automation to Our Advantage

A structured release and deployment process will automate many things that will improve the efficiency of the Salesforce Team, existing processes, and technology. It will reduce human error and make deployment operations easier. It will allow the team to spend more time on strategic thinking and less time on mundane deployment tasks

People-Centric

The Real Value of a structured release management process is that it helps improve the efficiency of people. By establishing a structured release process, we aim to naturally lead to better release efficiency and reduce turnaround time – creating best practices for collaboration and testing throughout the entire delivery lifecycle. As manual steps are greatly reduced, it paves way for a reduction in human error to a great extent thereby creating operational efficiency. This ultimately makes way for reliable release service quickly.

Tools

- Any Repository and version control tool. E.g.: GitHub, Bitbucket etc.
- CI Tool where we will write workflows for deployments to separate orgs e.g.: Bamboo, Circle CI, Travis CI, Jenkins, Copado etc
- Code quality tool – SonarQube, Checkmarks
- Test Automation Tools – Selenium, Cypress UI, Pro Var, Copado etc.

Branching Strategy

Need for Single Repository and Multiple Branching Strategies?

We have multiple Salesforce Environments. At any given point you could expect to see the following versions:

- Production – Version 2.
- Pre-Production – Version 2 (Always a copy of Prod commonly referred to as Staging).
- UAT: Version 2.1 (Customers are verifying changes about to go to production).
- QA / SIT: Version 2.2 (QA is testing the latest changes from the SIT or Previous sprint).
- Development: Version 2.3 (Latest user stories picked up for development).

Some use cases to think about:

- Development might push up to QA, but that is very unlikely.
- Typically, QA and SIT are typically testing the same version (Version 2.1).
- Occasionally, a release gets held up in UAT due to a customer concern, which requires extra bug fixes.
- A bug is found in production which requires a hotfix. It shouldn't step all over what is being tested in QA or UAT.
- A component or report created by business users directly in UAT or production has been overwritten because of a recent release.

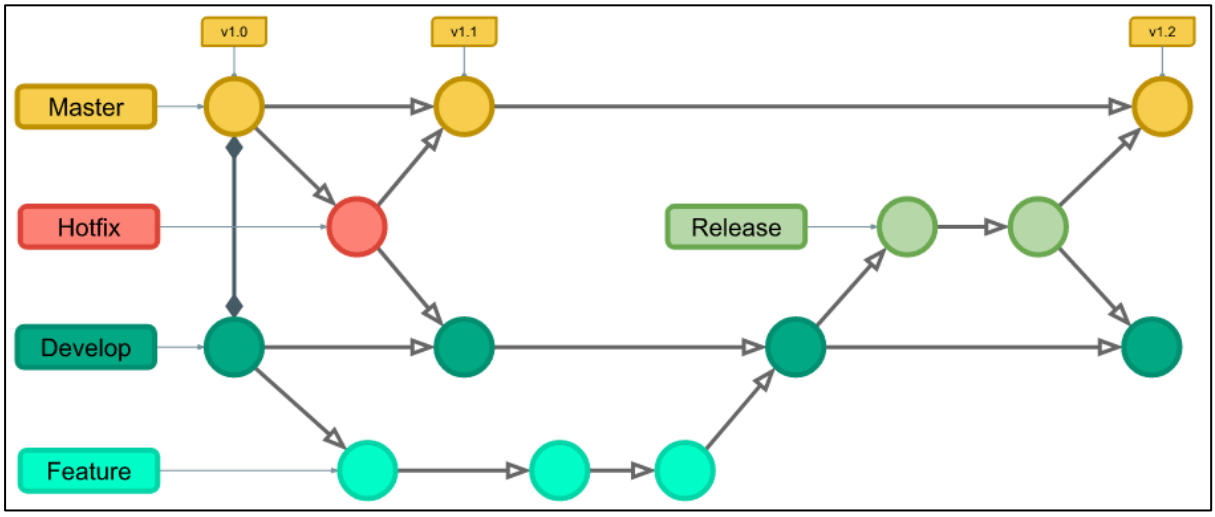
Proposed Branching Strategy

Our proposed workflow structure uses multiple different branches to record the history of the project. Each of these branches has a specific purpose and is used to isolate a part of the development pipeline for parallel processing and enhanced quality assurance. The master branch stores the official release history of Production, while the release branch serves as the main branch for performing deployments to UAT and Production features.

List of Branches:

Master, Hotfix, Release, Develop, Feature, and Backup branches

Backup Branch: There will be two backup branches – one each for UAT and Production code



Master Branch

The Master Branch will be the production codebase. The Salesforce team will ultimately merge all the production code into the Master branch after successful production deployment.

Release Branch

The Release Branch will be Salesforce UAT and Staging codebase. Think of it as almost a "pre-production" master branch. Develop and feature branches stem from this branch. The Release branch is also the branch from where Production deployment will happen before ultimately merging them with the master branch.

Hotfix Branch

The Hotfix Branch will be a replica of the Salesforce Production codebase. Hotfix branches are short-lived branches used exclusively to patch production code. Only critical production issues to the tune of P1 or P2 will be deployed from this branch. The idea is that by isolating problematic code from the Master or Release branch, you'll have more flexibility to quickly fix production critical bugs and test your fixes without affecting any of the scheduled UAT or the production releases directly (until you're ready to merge). Once a hotfix production deployment is successful, we must merge this fix back to the release branch for sync purposes.

Develop Branch

The Develop Branch will be Salesforce SIT and QA Codebase. The Develop Branch is used so you can avoid freezing the Release Branch while you prepare for a release—thus allowing teams *not* working on a specific release to continue their development work. The Develop branch is also the branch from where the UAT and SIT deployment will happen before ultimately merging them with the Release Branch in the UAT repository.

Feature Branch

Feature branches are where you'll work to develop new features—one branch per feature. These branches make it easy to track versioning on a more granular basis and to have various teams or developers working on multiple features at the same time. These feature branches will be created from Develop Branch. Once a developer has unit tested their code, they will create a pull request and merge it to develop branch. From Develop branch there will be a deployment done to SIT Org.

Backup Branch

There will be two Backup branches one each in Production and UAT Repository. As the name indicates, these will act as Back up branches for respective environments. The release manager would have to first run a backup job, that would back up the latest Production and UAT state into this backup branch. These branches are specifically created for rollback purposes. For now, Salesforce doesn't support rollback as efficiently. The only way is to redeploy the previous version of components as it was before. When a critical or high-priority bug is introduced or if a component is overwritten or deleted in production or UAT because of the latest release, this repository will act as a rollback repository.

Overall Development / Release Lifecycle

The Overall Development/Release lifecycle appears as listed below step by step:

1. A bug or a story feature is assigned to a developer from either a sprint or ticket list.

2. Developer will first create a feature branch from Develop branch in the Development repository. This is to ensure that the developer works on the most recent development codebase.

3. Developer will take a copy of this feature branch into the developer sandbox/scratch orgs where they will be working.

4. Once the initial code and unit test are done, the developer will put their new code in the feature branch

5. The next step is to create a pull request from the Feature branch and merge it into Develop branch.

Note: There may be code conflict and the tool should not allow merging code at this point. In case of code conflicts, the release manager will inform the developer. It is the responsibility

of the developer to ensure those conflicts are resolved and the latest code merged to the Develop branch

6. Developer will create and provide release notes to the release manager to perform deployment into Salesforce SIT Environment

7. Release Manager will create a manifest from the provided release notes and perform deployment from Develop branch into the SIT org and notify the developers and SIT Team of a successful or failed deployment.

8. Steps 4 to 7 will be repeated till a deployment is successful.

9. SIT Team will test the features and provide sign off to move the code into QA or UAT Phase.

10. For the features or user story signed off by the SIT Team, the developer will again provide release notes to the release manager to perform UAT deployment into Salesforce UAT Environment.

11. For the features or user story not signed off by the SIT Team, the developer will fix those issues and repeat 4 to 7 till the story is signed off and ready to deploy to UAT.

12. A true agile process does not wait for all user stories to be signed off to move to the next phase. So, there will always be circumstances where only a few features are signed off by SIT Team and the same needs to be pushed to UAT. In this scenario, the release Manager will first create a pull request from the Feature branch of the signed-off user stories and merge it into the Release branch in the UAT Repository.

Note: There may be code conflict and the tool shouldn't allow merging code at this point. In case of code conflicts, the release manager will inform the developer. It is the responsibility of the developer to ensure those conflicts are resolved and the latest code merged to the Release branch.

13. Release Manager will first run a backup job that will retrieve the entire salesforce UAT Org in the Backup branch.

14. Release Manager will collate all the release notes provided by different developers and create a manifest file and perform deployment from the Develop branch into the Salesforce UAT org and notify the developers and UAT Team of a successful or failed deployment.

15. If all the user stories are signed off by the SIT Team, the Release Manager will first create a pull request from Develop Branch and post successful UAT deployment will merge the same into the Release branch

16. If there arise circumstances where a component is overwritten in UAT causing critical issues or the UAT environment is rendered unusable because of the latest deployments, there may be a need to roll back deployment. Salesforce doesn't allow rollback, but we can always redeploy the earlier version of components back

17. Release Manager will re-use the manifest file created for UAT deployment and perform rollback deployment [redeploying previous version of components] again into the Salesforce

UAT org from the UAT Backup branch residing in the UAT Repository and notify the developers and UAT Team of a successful rollback.

18. UAT Team will test the features and provide sign off to move the code into Production.

19. For the features or user story signed off by the UAT Team, the developer will again provide release notes to the release manager to perform production deployment into Salesforce Production Environment.

20. For the features or user stories not signed off by the UAT Team, the developer will fix those issues and repeat 4 to 15 till the story is signed off and ready to be released to Production.

21. Release Manager will first run a backup job that will retrieve the entire salesforce Production Org in the Backup branch residing in the Production Repository.

22. Release Manager will collate all the release notes provided by different developers and create a manifest file and perform deployment from the Release branch into the Salesforce Production org. Post successful deployment, the release manager will create a pull request from the Release branch and merge the code to the Master branch in Production Repository. The release manager will notify the developers of a successful deployment.

23. If only a select few user stories are signed off by the UAT Team, Release Manager will create a temporary prod release branch from the release branch and give it release + today's date. In this scenario, the release Manager will first create a pull request from the Feature branch of the UAT signed-off user stories and merge it into the newly created temporary prod release branch residing in the UAT Repository.

24. Release Manager will collate all the release notes provided by different developers and create a manifest file and perform deployment from the newly created Prod Release branch into the Salesforce Production org. Post successful deployment, the release manager will create a pull request from the newly created prod release branch and merge the code to the Master branch in Production Repository. The release manager will notify the developers of a successful deployment.

25. If there arise circumstances where a component is overwritten in Production causing critical issues or the Production environment is rendered unusable because of the latest deployments, there may be a need to roll back deployment. Salesforce doesn't allow rollback, but we can always redeploy the earlier version of components back

26. Release Manager will re-use the manifest file created for Production deployment and perform rollback deployment [redeploying previous version of the same components] again into the Salesforce Production org from the Production Backup branch residing in the Production Repository and notify the developers of a successful rollback.

Notes for Developers

Few important points to note:

1) Before pushing any changes into the feature branch in source control, the developer should update their local repository in the <<Name of the Tool TBD (Ideally either source tree or VS Code)>> with the latest production code from Develop Branch.

2) After updating their local repository, they will Commit and Push their changes from the Local repository to the 'feature or bugfix' branch.

3) During the creation of a pull request if there are any code conflict errors then the Release Manager will inform the same to developers who will have to resolve this code conflict and push the updated code into the repository.

4) It is primarily the responsibility of developers to ensure that Release Notes are produced and shared with Release Manager in advance of the deployment schedule.

5) Though the release manager will compare and merge the code, please note that it is primarily the responsibility of the developer to ensure that all the components in release notes match that of components in the feature branch and resolve conflicts if any.

6) If you need to delete any component as part of pre-Deployment activity, then add those members to src/destructiveChangesPre.xml.

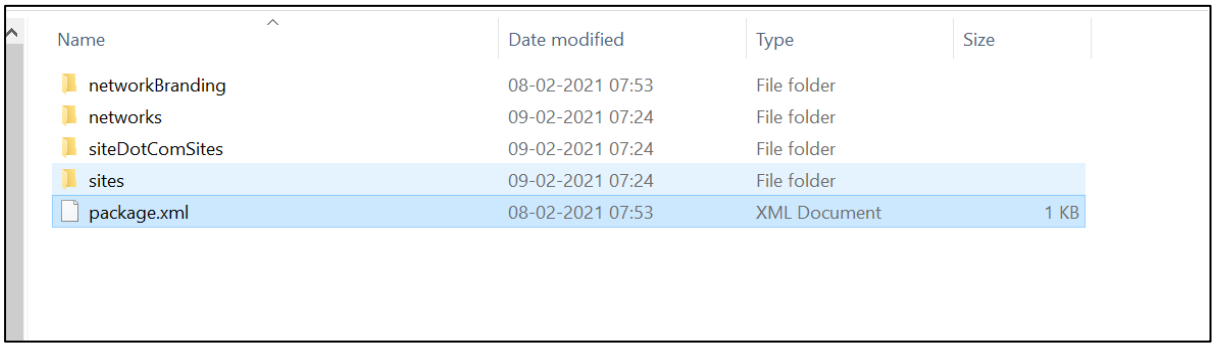
7) If you need to delete any component as part of a post-Deployment activity, then add those members to src/destructiveChangesPost.xml.

8) Compare your changes by using any comparator tool<<Name of the tool TBD (Ideally KDIF)>>. You need to add all your changes from your localSandboxRepository to LocalBranchRepository.

- If you have created new labels, then add your labels and don't remove anything from the label file.
- Be more careful if you have made changes to your profile. Don't override any existing changes. Just add your changes.
- If you need to create a queue against a custom object, the custom object must be migrated first.
- Run local tests in a deployment to a development environment, such as a sandbox, by setting the RunLocalTests test level in the deployment options. Running tests in a development environment allows you to catch and fix any failures early and ensure a better deployment experience for UAT and Production.
- Update the manifest file with your changes. All your components should be available inside build/project-manifest-ProjectName.txt.

Community Deployment:

- For community deployment no manifest is needed.
- Most of the community changes are stored in the following four files/ folders – Network branding, Network, Sitedotcomsites and sites.



Name	Date modified	Type	Size
networkBranding	08-02-2021 07:53	File folder	
networks	09-02-2021 07:24	File folder	
siteDotComSites	09-02-2021 07:24	File folder	
sites	09-02-2021 07:24	File folder	
package.xml	08-02-2021 07:53	XML Document	1 KB

- The below should be the corresponding package.xml while retrieving/deploying the community

```
<?xml version="1.0" encoding="UTF-8"?>
<Package xmlns="http://soap.sforce.com/2006/04/metadata">
    <types>
        <members>xyzsite</members>
        <name>CustomSite</name>
    </types>
    <types>
        <members>xyzsite</members>
        <name>Network</name>
    </types>
    <types>
        <members>xyzsite1</members>
        <name>SiteDotCom</name>
    </types>
    <types>
        <members>xyzsite Profile</members>
        <name>Profile</name>
    </types>
    <version>30.0</version>
</Package>
```

- Below changes are always manual in the community and the developer should give clear step by step instructions under the section Manual Community changes in the release notes.
    - Audience assignment
    - User Profiles Label changes
    - Component Re-ordering
    - Component Re-naming
    - Page Variation Changes
    - Setting up Priorities

Conflict Resolution

Sometimes while merging code from one branch to another, we can get conflicts. Source Control will not allow us to merge code in case of conflicts and the same needs to be resolved.

1. Introduction: The Inevitable Friction in the DevOps Revolution

1.2 Why Conflicts Arise in DevOps

- The inherent differences in traditional Dev and Ops mindsets (e.g., speed vs. stability).
- The rapid pace of change and constant integration.
- Interdependencies and shared responsibilities.
- The human element: personality clashes, communication styles.

1.3 The Cost of Unresolved Conflict

- Project delays and missed deadlines.
- Reduced team morale and increased burnout.
- Decreased productivity and quality.
- Blame culture and erosion of trust.
- Employee turnover.

2. Common Sources and Types of DevOps Conflicts

2.1 Cultural and Mindset Clashes

- **"Move Fast and Break Things" vs. "Maintain Stability at All Costs":** The core philosophical divide.
- **Ownership and Responsibility:** Who owns the pipeline? Who is responsible for an outage?
- **Risk Aversion vs. Innovation:** Operations' natural tendency towards stability vs. Development's drive for new features.
- **Tooling Preferences:** Disagreements over choice of CI/CD tools, monitoring systems, etc.

2.2 Process and Workflow Disagreements

- **Deployment Cadence:** How often should deployments occur? (e.g., daily vs. weekly).
- **Release Management:** Definition of "done," testing gates, rollback procedures.
- **Change Management:** How changes are approved, documented, and communicated.
- **Incident Response:** Roles, responsibilities, and communication during outages.

2.3 Communication Breakdowns

- **Lack of Transparency:** Information silos between Dev and Ops.
- **Ambiguous Requirements:** Misinterpretations of user stories or operational needs.

- **Ineffective Feedback Loops:** Lack of constructive criticism or inability to share insights effectively.
- **Personality Clashes:** Different communication styles, directness vs. diplomacy.

2.4 Technical and Resource Conflicts

- **Resource Contention:** Shared infrastructure, budget limitations.
- **Technical Debt Prioritization:** Balancing new features with refactoring and bug fixes.
- **Performance vs. Features:** Disagreements on optimization efforts.
- **Security Concerns:** Balancing rapid development with robust security protocols.

3. Frameworks and Strategies for Proactive Conflict Prevention

3.1 Fostering a Culture of Blamelessness and Psychological Safety

- **Post-Mortems/Blameless Retrospectives:** Focus on process improvement, not individual blame.
- **Promoting Openness and Vulnerability:** Encouraging team members to admit mistakes and seek help.
- **Leadership by Example:** Leaders demonstrating transparency and humility.

3.2 Establishing Shared Goals and Metrics

- **OKRs (Objectives and Key Results):** Aligning Dev and Ops teams on common business objectives.
- **DORA Metrics (Deployment Frequency, Lead Time for Changes, Mean Time to Restore, Change Failure Rate):** Shared understanding of performance and areas for improvement.
- **Cross-Functional KPIs:** Metrics that require collaboration to achieve.

3.3 Enhancing Communication and Collaboration Mechanisms

- **Dedicated Communication Channels:** Slack, Microsoft Teams, etc., for real-time interaction.
- **Regular Stand-ups and Cross-Team Meetings:** Facilitating information exchange.
- **"Definition of Done" and Clear Handoffs:** Minimizing ambiguity between stages.
- **Pair Programming and Cross-Training:** Fostering empathy and understanding of different roles.
- **Documentation and Knowledge Sharing:** Centralized repositories for policies, procedures, and best practices.

3.4 Implementing Robust Processes and Automation

- **Version Control for Everything (Infrastructure as Code, Configuration as Code):** Reducing manual errors and fostering transparency.

- **Automated Testing and Deployment Pipelines:** Shifting left, catching issues early.
- **Standardized Environments:** Minimizing "it works on my machine" issues.
- **Clear Escalation Paths:** Defined procedures for addressing issues that cannot be resolved within the team.

4. Effective Conflict Resolution Techniques

4.1 Early Detection and Intervention

- **Active Listening:** Understanding the underlying concerns, not just the stated position.
- **Observing Non-Verbal Cues:** Recognizing tension or discomfort.
- **Regular Check-ins:** Proactively addressing potential friction points.

4.2 Collaborative Problem-Solving

- **Win-Win Approach (Integrative Bargaining):** Finding solutions that satisfy both parties' needs.
- **Brainstorming Sessions:** Generating multiple solutions before evaluating.
- **Root Cause Analysis (e.g., 5 Whys, Fishbone Diagram):** Getting to the heart of the issue.
- **Focus on Interests, Not Positions:** Understanding *why* someone holds a particular viewpoint.

4.3 Mediation and Facilitation

- **Role of a Neutral Third Party:** Someone unbiased to guide the discussion.
- **Setting Ground Rules:** Ensuring respectful and productive dialogue.
- **Summarizing and Clarifying:** Ensuring everyone understands each other's perspectives.
- **Facilitating Agreement:** Helping parties reach a mutually acceptable resolution.

4.4 Negotiation Strategies

- **BATNA (Best Alternative to a Negotiated Agreement):** Knowing your fallback position.
- **Principled Negotiation:** Separating the people from the problem, focusing on objective criteria.
- **Active Empathy:** Stepping into the other person's shoes.

4.5 When to Escalate

- **Clear Escalation Matrix:** Defined steps and responsible parties.
- **Documentation of Attempts:** What was tried, what was the outcome.
- **Involving Management:** For intractable issues or those impacting broader organizational goals.

5. Case Studies/Examples (Conceptual)

Case Study 1: The "Release Train vs. Hotfix" Showdown

- *Problem:* Dev wants frequent small releases, Ops needs stability for large systems.
- *Resolution:* Implemented automated testing, canary deployments, and a clear hotfix process with shared ownership.

Case Study 2: The "Tooling Tug-of-War"

- *Problem:* Dev wants modern CI/CD, Ops prefers legacy, battle-tested tools.
- *Resolution:* Piloted new tools on non-critical projects, established a "Tooling Guild" for evaluation, and focused on integration rather than replacement.

Case Study 3: The "Blame Game" After an Outage

- *Problem:* Finger-pointing between Dev and Ops after a production incident.
- *Resolution:* Instituted blameless post-mortems, shared responsibility for MTTR, and cross-functional incident response training.

6. Metrics for Success and Continuous Improvement

6.1 Measuring the Impact of Conflict Resolution Efforts

- Reduced incident frequency and severity.
- Improved deployment frequency and lead time.
- Higher team satisfaction and retention (e.g., through surveys).
- Faster problem-solving time.
- Increased innovation and experimentation.

6.2 Feedback Loops and Iteration

- Regular retrospectives on conflict resolution processes themselves.
- Surveys and interviews to gather team sentiment.
- Adjusting strategies based on outcomes.

6.3 Leadership's Role in Sustaining a Collaborative Culture

- Continual reinforcement of shared values.
- Investing in training and development.
- Recognizing and rewarding collaborative behaviours.

7. Conclusion: Building Resilient and High-Performing DevOps Teams

- Recap of the main arguments: Conflicts are opportunities for growth.
- Emphasize that effective conflict resolution is a continuous journey, not a destination.
- Reinforce the benefits: Stronger teams, better products, more satisfied customers.
- Final call to action: Invest in people, processes, and tools to foster a truly collaborative DevOps environment.

-: END :-