Release Management Guidelines

CRM Release Management guide can be found at our Wiki, please make sure that you read the information available as well to understand the overall environment strategy, org strategy and follow the process.

Responsibilities in Salesforce Development and Release Validation

When engaging in any development activity in Salesforce, developers are responsible for ensuring the quality and integrity of the code by following these key practices.

- **Execution of Test Classes:**
  If a developer is making changes to any existing code, the developer needs to ensure that he runs the associated test classes before he starts the development to ensure all the existing code is working as desired. And once he is done with his development the developer must execute all related test classes associated with the development again.This step is crucial to verify that the new code or changes do not introduce any issues into the existing system.

- **Error Resolution:**
  If any errors are encountered during the execution of these test classes post his development, it is the developer's responsibility to investigate and fix those errors promptly. This ensures that the code functions as expected and maintains system stability.

- **Release Validation:**
  In the event of any errors encountered during the release validation process, it is the responsibility of the developer who is accountable for the specific development activity to address and resolve the errors. The developer must ensure that the release is error-free before it is finalised and deployed. The DevOps team will be sharing the build failure information in the common chat group (To be created for the intended use for Deployment communication and collaboration) which will have all the developers, tech leads, architects associated with that release. DevOps team will be tagging the developer responsible post which developer needs to fix the build error and update the same on the same chat group.

- **Code Review:**
  Role : Developer

Primary goal: Deliver working code with minimal downstream failures and review churn.

Responsibilities:

- Complete development plus Unit Testing (UT) in the dev environment.
- Pre-validate changes against the Quality Assurance (QA) branch before opening a Pull Request (PR) to:
  - Prevent avoidable Continuous Integration (CI) failures,
  - Protect shared environments
  - Improve release/merge throughput
  - Reduce review churn so reviewers spend time on design/quality, not break-fix.
- Raise PR for Tech Lead review once validation passes.

Role : Tech Lead

Primary goal: Ensure the code is merge-ready, maintainable, and aligned to standards.

Approval model:

- If the change is Out-of-the-Box (OOTB) with no customization/complex logic Tech Lead approval is sufficient to merge upward.
- If the change is customized/complex feature work Tech Lead provides Approval #1, then routes for Technical Architect Approval #2.

Responsibilities:

- Confirm the implementation follows the coding standards defined in your referenced document.
- Verify functional completeness and quality: key scenarios covered
- Test classes working and with “proper defined coverage” (as per standards
- No obvious regression risks.
- Ensure any required documentation is complete, accurate, and kept with the right artifacts (e.g., Pre and Post Deployment Steps).
- Ensure that if for any reason (e.g. timeline constraints) guidelines are not followed, Syensqo MUST be informed and a technical debt created.

Role : Technical Architect

Primary goal: Protect architecture integrity and long-term scalability for higher-risk changes.

When involved: Any PR with customization, complex logic, or complex feature behavior requiring two approvals. Also he can do periodic reviews just to ensure that best practices are being followed in other non-complex and non-customized implementations also.

Responsibilities:

- Perform architectural and design validation beyond line-by-line code review, alignment with platform/reference architecture, correct patterns and boundaries (e.g., layering, dependency direction),
- Non-functional requirements (performance, security, reliability), backward compatibility and extensibility.
- Confirm the change can be merged with higher environments from a system-impact perspective.

- **Accountability:**
  Each developer is directly accountable for the development tasks they work on. If an error arises as a result of their development during the release process, they must take ownership of fixing the error. Collaboration with relevant teams may be required to ensure the issue is resolved in a timely manner.

- **Pre/Post Deployment Step Manual Tasks:** Developers are responsible for clearly detailing the manual tasks required in the Jira ticket respective fields, including the time estimation for the execution of these, so the release manager can estimate the total estimated time of manual work per release deployment. Manual tasks must be written in a way that a Release Manager without context of each ticket can understand step by step what has to be done.

In practice

In best practice, creating feature branches at the start of development ensures your work is isolated from the main codebase, preventing incomplete changes from affecting the main branch. It allows multiple developers to work on different features simultaneously without conflicts and provides a clear context for tracking progress and collaboration. This approach also enables early testing and continuous integration, helping catch issues early and making it easier to merge changes smoothly into the main branch.]

Step 1: Checkout and refresh integration branch

Before starting work on a new feature, create a feature branch from the integration branch (develop) using this naming convention feature/US-XXXX; replace XXXX` with your User Story or task ID. Make sure you are on the integration branch to start with and you pull the latest from your remote before creating the branch.

git fetch --all prune

git checkout develop

git pull origin develop

git checkout -b feature/US-123

- Branch Types:
  - Feature: feature/US-XXXX
  - Bugfix: bugfix/US-XXXX
  - Hotfix: hotfix/US-XXXX

Step 2: Ready to develop

You can now make your change locally (e.g.: classes) or in your dev sandbox. Once ready pull the latest changes from Salesforce using sf command.

sf project retrieve start -o myDevOrg -m "CustomObject:Account"

Step 3: Stage and Commit your changes

git add force-app/main/default/objects/Account

git commit -m "[US-123] My Account changes"

git push origin feature/US-123

You can repeat steps 3 & 4 until changes are complete and the feature is tested.

Validate Feature

After completing your development and testing locally, it's crucial to ensure that the feature works as expected in a collaborative environment(QA). This involves merging your changes back into the integration branch(develop), where they can be further tested and validated by the team. Follow these steps to validate your feature:

Step 4: Create a Pull Request (PR)

Open a PR to merge into the integration branch (develop). Consult your tech lead if you are unsure of the target branch.

- Pipeline Build Steps:
  - Static Code Analysis: Automated with Apex PMD.
  - Validation Build: Ensures code quality.
  - Apex Tests: Runs only relevant tests.
- Inspect Build Reports:
  - Tests: View failed tests.
  - HTML Viewer: See static code analysis results.
  - Code Coverage: Review coverage for changed classes.

    - Validation Build: Check reports via the PR build link.
    - Key Tabs:
- Handle Build Failures: If the build fails, fix issues before requesting a PR review.
- Review Process: If your build is successful, your reviewer will review and merge the PR into develop.

How to write a good pull request ?

A good pull request (PR) can make or break your code review game and streamline your project's integration process. Here's how to craft a PR that stands out and gets the job done:

1. **Punchy Title and Insightful Description**

- Title: Start with the user story number and a brief summary. For example, "[US-123] Fix login page authentication bug" or "[US-456] Add user profile feature."
- Description: Dive deep into what's changed with:
    - Context: Explain why these changes matter—whether it's a bug fix, a shiny new feature, or some refactoring.
    - Changes Made: Give a quick rundown of what's new, what's improved, or what's been removed.
    - Related Issues/Tickets: Link to any relevant issues or tickets so reviewers can dive deeper if needed.
    - Testing: Outline how you've tested the changes or how they should be tested. If needed, add test cases or steps.
    - Screenshots or Examples: For visual changes, throw in some screenshots or GIFs to showcase the updates.

1. **Code Quality and Organization**

- Clean Code: Make sure your code is polished—no stray comments, debug lines, or formatting issues. Stick to project coding standards.
- Commits: Keep commits focused and meaningful. Each commit should handle one thing at a time.

1. **Documentation**

- Code Comments: Comment on tricky parts of the code to help others (and your future self) understand your logic.
- Documentation Updates: If your PR impacts documentation, update the relevant docs or note it in the PR description.

1. **Issue Tracking**

- Issue Reference: Link to the related issue or work item in Jira, i.e.: US-123. .

1. **Approval and Sign-Off**

- Functional Approval: Ensure all necessary reviews and functional approvals are in place before merging. As once merged, this will automatically deploy your components to the next inline environments (i.e.: QA).
- Sign-Off: If your project requires formal sign-off, make sure to get it from the required team members or stakeholders.

1. **Review Requests**

- The github pipeline will enforce certain approval gates for quality controls. Make sure everything is clean before you reach out to your tech lead.
- Reviewers: In the pull request, tag the right Teach Lead reviewers—those who know the code or can offer valuable insights.
- Any pull request that only contains declarative changes, then only a review from tech lead is enough. Otherwise if the pull request contains any code like apex or lwc, then will need proper rieview from an architect (i.e.: David, Stephane).
- Review Notes: Point out any specific areas where you need feedback.

1. **Merge Considerations**

- Merge Conflicts: Check for conflicts before requesting a review. Resolve any issues to ensure a smooth merge.

Example of a Pull Request

Title: [US-789] Add user profile feature

Description:
This PR introduces a new user profile feature, enabling users to view and edit their profile information. Here's a breakdown:

- Added a UserProfile component for displaying user details.
- Implemented profile editing with validation.
- Updated the user settings page to include profile management.

Related Issues: Closes [US-789]

Testing:

- Verified correct loading of the profile page and display of user data.
- Tested various scenarios for profile editing functionality.
- Confirmed that all tests are passing.

Tools

Tool Name	Used for	Used By

GitHub Enterprise	The "Where" of the Code. GitHub is a cloud-based platform for version control and collaboration using Git. The "Enterprise" version is specifically designed for large organizations, offering enhanced security (like SAML Single Sign-On), centralized administration, and the ability to self-host on private servers.
	• Primary Use: Storing and managing source code.
	• Key Features: * Repositories: Where the code "lives."
	  ◦ Pull Requests (PRs): A process for developers to review each other's code before merging it into the main project.
	  ◦ GitHub Actions: Automated pipelines for testing and deploying code (CI/CD).
	  ◦ Security: Advanced secret scanning and vulnerability alerts.
		Tech Lead
		
		Developers

Jira	The "What" and "When" of the Work. Developed by Atlassian, Jira is the industry standard for Agile project management. It is used to track "Issues"—which can be anything from a small bug to a massive new feature.
	• Primary Use: Planning, tracking, and managing software projects.
	• Key Features:
	  ◦ Boards (Scrum/Kanban): Visual columns (To Do, In Progress, Done) that show the status of tasks.
	  ◦ Tickets/Issues: Individual task cards containing descriptions, assignees, and due dates.
	  ◦ Backlogs: A prioritized list of work to be done in the future.
	  ◦ Reporting: Burndown charts and velocity reports to see how fast the team is working.
		All

Confluence	The "Why" and "How" of the Strategy. Also an Atlassian product, Confluence is a corporate wiki or knowledge-management platform. While Jira is for tasks, Confluence is for information.
	• Primary Use: Documentation and team collaboration.
	• Key Features:
	  ◦ Pages & Spaces: Folders (Spaces) that contain documents (Pages) ranging from meeting notes to technical specs.
	  ◦ Templates: Pre-built layouts for Project Plans, Architecture Decisions, and Requirements.
	  ◦ Real-time Editing: Similar to Google Docs, but structured specifically for technical teams.
		All

DataLoader	Data Loader is a powerful tool used for importing, exporting, and deleting data in Salesforce. Data Loader supports Data Migration, Data Import, Data Export, Scheduled Data Loads	All
Visual Studio	Visual Studio is an integrated development environment (IDE).It's primarily used for software development, Here are some common use cases for Visual Studio ,Software Development,Code Editing,Debugging ,Version Control Integration ,Project Management,Code Analysis and Testing,Extensions and Customization	Developers
Workbench	Workbench is a powerful web-based tool provided by Salesforce that allows administrators and developers to interact with Salesforce organizations through the Salesforce APIs. It offers a variety of features and functionalities that aid in Salesforce administration, development, and debugging	Administrators /Developers /ReleaseManagers

Communication Channels

<this section aims to include all communication channels, for example Jira comments, and google/microsoft team groups, first with the goal to support project, then after to be adjusted for maintenance>

Github Subscriptions

To integrate GitHub with Microsoft Teams, you'll primarily be using the GitHub for Teams app. This allows your team to stay updated on pull requests, issues, and deployments directly within your Teams channels.

Installation Steps:

1. Open Microsoft Teams: Go to the Apps icon in the bottom-left corner of the Teams sidebar.
2. Search for GitHub: Type "GitHub" into the search bar. Select the official GitHub app.
3. Add to a Team: Click the dropdown arrow next to "Add" and select Add to a team. Search for the specific channel where you want the notifications to appear.
4. Set Up Notifications: Once added, the GitHub bot will appear in the channel.
5. Sign In: Type @github signin in the message box. A window will pop up asking you to authorize the connection between your GitHub account and Microsoft Teams.

Basic Commands to Connect Repositories

Once installed, you need to "subscribe" to specific repositories to receive updates. Use these commands in the channel:

- Subscribe to a Repo: @github subscribe owner/repository-name
- Unsubscribe: @github unsubscribe owner/repository-name
- List Subscriptions: @github subscribe list