Chapter 1: The Why, When and By Whom

This chapter explores the fundamental considerations of Salesforce deployments within the context of consulting projects. It addresses:

- **Why Deploy**: The importance and benefits of deploying Salesforce metadata throughout the project lifecycle, from the build phase to UAT to GoLive.
- **When**: When in the project timeline should deployments be planned and executed to ensure smooth progress and mitigate risks. (Hint - it's often, but not in every org)
- **By Whom**: Roles and responsibilities involved in the deployment process, such as consultants committing changes, architects reviewing commits and system elements, and release managers overseeing and executing deployments.

Why do I Deploy ?

In traditional software development, deployments often occur to migrate changes between environments for testing or production releases. However, in the context of Continuous Integration (CI) and Salesforce development, deployments are just synchronization checkpoints for the application, irrelevant of the organization.

Said differently, in CI/CD Deployments are just a way to push commits to the environments that require them.

CI deployments are frequent, automated, and tied closely to the development cycle.

Deployments are never the focus in CI/CD, and what is important is instead the commits and the way that they tie into the project management - ideally into a ticket for each commit.

In software development, a **commit** is the action of saving changes to a version-controlled repository. It captures specific modifications to files, accompanied by a descriptive message. Commits are atomic, meaning changes are applied together as a single unit, ensuring version control, traceability of changes, and collaboration among team members.

Commits are part of using Git.
Git is a distributed version control system used to track changes in source code during software development. It is free and widely used, within Salesforce and elsewhere.

So if deployments are just here to sync commits...

Why do I commit ?

> As soon as a commit is useful, or whenever a day has ended.

Commits should pretty much be done "as soon as they are useful", which often means you have fulfilled **one** of the following conditions:

- you have finished working a ticket;
- you have finished configuring or coding a self-contained logic, business domain, or functional domain;
- you have finished correcting something that you want to be able to revert easily;
- you have finished a hotfix;
- you have finished a feature.

This will allow you to pull your changes from the org, commit your changes referencing the ticket number in the Commit Message, and then push to the repository.
This will allow others to work on the same repository without issues and to easily find and revert changes if required.

You should also commit to your local repository whenever the day ends - in any case you can squash those commits together when you merge back to Main, so trying to delay commits is generally a bad idea.

Take the Salesforce-built "Devops Center" for example.

They tie every commit to a Work Item and allow you to chose which elements from the metadata should be added to the commit. They then ask you to add a quick description and you're done.
This is the same logic we apply to tickets in the above description.

> If you're wondering "why not just use DevOps Center", the answer is generally "you definitely should if you can, but you sometimes can't because it is proprietary and it has limitations you can't work around".
> Also because if you learn how to use the CLI, you'll realise pretty fast that it goes WAY faster than DevOps Center.

To tie back to our introduction - this forces a division of work into Work Items, Tickets, or whatever other Agile-ism you use internally, and the project management level.

> **DevOps makes sense when you work iteratively, probably in sprints, and when the work to be delivered is well defined and packaged.**

This is because....

When do I Deploy ?

Pretty much all the time, but not **everywhere.**

In Salesforce CI/CD, the two main points of complexity in your existing pipeline are going to be:

- The first integration of a commit into the pipeline
- The merging of multiple commits, especially if you have the unfortunate situation where multiple people work in the same org.

The reasons for this are similar but different.

In the case of the first integration of a commit into the pipeline, most of the time, things should be completely fine. The problem is one that everyone in the Salesforce space knows very well. The Metadata API **sucks**. And sadly, SFDX... also isn't perfect.
So sometimes, you might do everything right, but the MDAPI will throw some file or some setting that while valid in output, is invalid in input. Meaning Salesforce happily gives you something you can't deploy.
If this happens, you will get an error when you first try to integrate your commit to an org. This is why some pre-merge checks ensure that the commit you did can be deployed back to the org.

In the case of merging multiple commits, the reasons is **also** that the Metadata API **sucks.** It will answer the same calls with metadata that is not ordered the same way within the same file, which will lead Git to think there's tons-o-changes... Except not really. This is mostly fine as long as you don't have to merge your work with someone else's where they worked on the same piece of metadata - if so, there is a non-zero chance that the automated merging will fail.

In both cases, the answer is "ask your senior how to solve this if the pipeline errors out". In both cases also, the pipeline should be setup to cover these cases and error out gracefully.

"What does that have to do with when I deploy? Like didn't you get lost somewhere?"

> The relation is simple - you should deploy pretty much ASAP to your remote repo, and merge frequently to the main work repository. You should also pull the remote work frequently to ensure you are in sync with others.
> Deploying to remote will run the integration checks to ensure things can be merged, and merging will allow others to see your work. Pulling the other's work will ensure you don't overwrite stuff.

> Deploying to QA or UAT should be something tied to the project management cycle and is not up to an individual contributor.
> For example, you can deploy to QA every sprint end, and deploy to UAT once EPICs are flagged as ready for UAT (a manual step).

Who Deploys ?

Different people across the lifecycle of the project.

On project setup, the DevOps engineer that sets up the pipeline should deploy and setup.
For standard work, you should deploy to your own repo, and the automated system should merge to common if all's good.
For end of sprints, the automated pipeline should deploy to QA.
For UAT, the Architect assigned to the project should run the required pipelines.

In most cases, the runs should be automatic, and key points should be covered by technical people.

Revision #3
Created 18 June 2024 14:35:09 by Windyo
Updated 11 March 2025 09:20:47 by Windyo