A mature Development and Deployment Process

How to sleep well when your 200 developers deploy apps every day?

Author
Sascha Wildgrube, Principal Technical Consultant, ServiceNow

Table of Contents

Disclaimer

This document is NOT released or published by ServiceNow. It does in no way express official ServiceNow's opinions nor shall any of its contents be interpreted as a statement made by ServiceNow.

This document expresses the individual opinions of its author which are based on observations made and work conducted with and for a limited number of ServiceNow customers.

Neither the author nor ServiceNow shall be liable for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising from following the guidance contained in this document.

This version of this document should be considered as work in progress. It is knowingly incomplete and subject to change and refinement.

License

Introduction

Who should read this Document?

This document is for

- Platform Owners
- Architects
- Development Team Leaders
- Developers
- Everyone who wants to modernize their development and deployment process
- Everyone who consults and guides organizations on the topic
- Anyone who suffers from sleep deprivation caused by deployments that have gone terribly wrong

Purpose of this Document

This document aims to provide specific guidance on how to introduce and establish a robust, reliable, fast, and automated development and deployment process.

It contains guidance on the process, the techniques required to make it work on a technical level, the technical capabilities required (which go beyond OOTB), a proposed training schedule and on how to introduce the process into the organization.

This document proposes three different variants of instance setups – and further guidance on how to choose from these variants.

The readers can use it as the foundation for a transformation project that takes the organization from where it is today towards a path of process maturity – for new and long existing ServiceNow instances.

It should be noted however that the proposed process is ONE way to solve a specific set of problems. It is not the ONLY way.

With that being stated, the purpose of this document is not only to help organizations to solve problems but also to inspire and contribute to the discussion on how application development and deployment should be supported in the ServiceNow Platform in the future.

The Problem – Increasing Complexity

The reasons why you are reading this might vary. Maybe there are specific problems to be addressed, maybe there is just a foggy sense of things just do not work out as they should and maybe you are just interested.

Here is a list of observations, the organization might have made:

- Long time-to-market / rare and huge releases
- Many roundtrips between test and development
- Time-consuming manual tests
- As more customizations are introduced over time, side-effects are more difficult to foresee – resulting in less accurate estimates despite increasing team experience
- Difficulties to respond to last-minute Product Owner decisions
- Difficulty to track the exact status of "story" delivery
- Inconsistency between development, test, and production instances
- Deployments don't produce the exact state in test environments as delivered (or intended) by the development team
- Deployments don't produce the exact state in production as it was previously tested
- Incident-driven fixes (in production) don't make it back into development – and reappear after subsequent releases
- Talented and experienced team members leave and seek other opportunities quickly

The last bullet point is a tricky one. The observation might have been made, but the link to the existing development and deployment process might not yet have been established. Someone leaving the team has always multiple reasons and it is impossible to blame one single factor. The author of this document can't provide any evidence of causality but believe that process maturity is a factor for staff acquisition and retention.

The Solution - From Update Sets to Source Control

The Paris release was a game-changer. Source-control based application installations became delta-aware. This means, that if an application already exists on an instance and a new version of that same application is installed through source-control, existing database tables and fields are preserved. Prior to Paris, an application was completely uninstalled, and all data deleted, then the new version was installed – which was ok for sub-production (if a proper test data creation process was in place) – but obviously not ok for production.

Paris pushed the door wide open for source-control-based deployments on production.
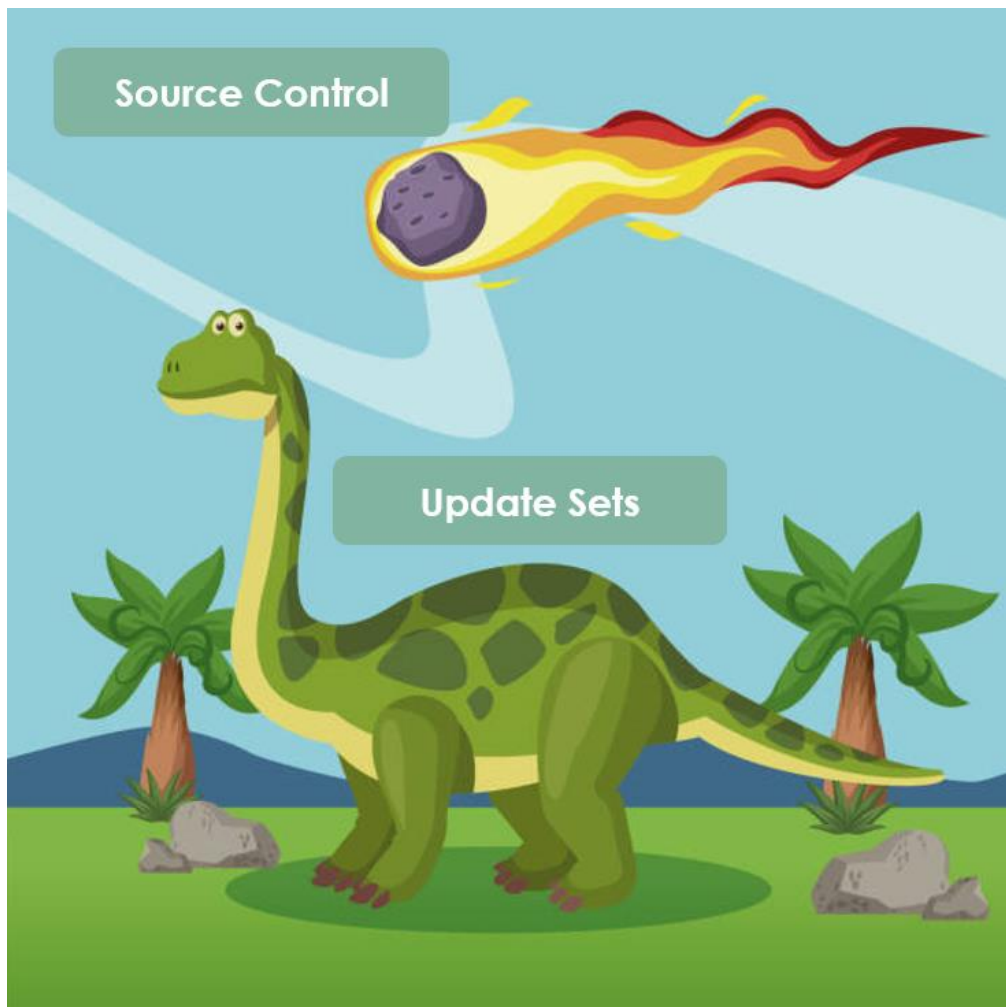
If ServiceNow was a planet, the Paris release was to Update Sets, what the meteor hitting earth about 66 million years ago was to dinosaurs – an extinction event.

Current scientific research indicates that the dinosaurs did not die from the impact itself – the process took decades – and so Update Sets are still in action in many if not most ServiceNow instances.

The technical capability alone doesn't drive change. Many people, with their belief systems, their routines, their habits, and comfort zones are inclined to ignore these opportunities as long as they can.

The solution proposed in this document may appear radical. It requires unlearning quite a few things, it requires adopting new habits, learning new techniques.

The three guiding principles:

- Always deploy applications – instead of selected records captured in Update Sets
- Zero-tolerance mind-set regarding failed tests or Instance Scan findings
- Application-centric architecture instead of individual customizations

If you cannot buy into these principles personally, or you believe that the organization you are working for will never ever adopt these principles, you might just stop reading here. In that case, the guidance you will find in this document is not for you or your organization.

Terminology

A few notes on terminology used in this document:

- "Application" refers to a scoped or global application represented by a sys_app record in the ServiceNow instance
- "Backlog item" is a statement of demand, usually represented by a single record in a ticket system (e.g., a "Jira issue", a "Story", a card on a Kanban board or something similar)
- "Release Package" is a specific set of application versions to be deployed on a target instance including any additional information required to perform the deployment
- "Dependency" refers to the relationship between two applications where one application depends on another when the first application in any form or shape needs the other application to function or even to be installed in the first place.
- "Baselining" is the process of creating a version from the dev branch of an application at a specific point in time. This "Baselined Application Version" will then remain static and unchanged forever.

What to expect from this Document

This document outlines the process as a whole and describes 6 quality gates – each with a checklist to work on, references to the relevant building blocks and recommendations on who could act as a gatekeeper.

The 13 building blocks described in this document are either documents, detailed checklists, procedures, technical capabilities, additional systems to set up or a combination of these.

Two of the main building blocks are based on automated quality control – the verification against the Coding Guideline and the use of Automated Tests are instrumental. These and all other testing activities are part of quality control activities, which are described in detail chapter "Testing".

3 variants of instance setups are described.

The document describes techniques that the team should adopt to be able to execute the process.

The document also describes practical advice on how to introduce the process and to manage the organizational change and enablement.

A note on Open-Source Applications

Some of the proposed techniques require technical capabilities that are not available OOTB as of the Washington DC release. There are some open-source applications available that can be used to fill in the gaps.

References to these open-source applications are made in various chapters in this document.

The process and the techniques described in this document can be implemented and adopted with or without using the open-source applications referenced in this document. These applications have been developed to support the process and the techniques. Re-implementing comparable capabilities on top of the platform will require some time and effort.

Of course, it is at the organization's discretion whether these applications are installed and used or extended or re-implemented and used as inspiration.

A key element to the process is pipeline automation. The Deployer open-source application supports a fully automated deployment process. Countless developer hours and real-world deployment experience made it what it is today. A make-or-buy decision should not be made unless the Deployer app has been evaluated.

Process Overview

As for every process, it requires three factors to play all well together: people, technology, and organization.

In the following chapters we will provide a high-altitude overview on the process, describe in detail what the building blocks for the process are, which quality gates and procedures need to be put in place and which techniques the development teams need to learn and apply.

Further, we will outline an approach to introduce the process into an existing or new organization.

When thinking in terms of a process, the most obvious question is: what represents one single instance of the process?

The process ranges from end-to-end: from the documentation of an idea or demand stated by a Product Owner representing the business to the moment in time when a new or changed application capability can be used in production.

The process has 3 phases:

- Implementation
- Application Delivery
- Test and Production Deployment

The process "instance" is a different one in each of the phases:

- During "Implementation":
  A single backlog item
- During "Application Delivery":
  An application version
  (which embodies changes resulting from one or more backlog items)
- During "Test and Production Deployment":
  A release package
  (which is a combination of one or more application versions and instructions on how to install them on a target instance)

The process has 6 quality gates:

- During Implementation
  - Backlog item is "Ready"
  - Backlog item is "Done"
- During Application Delivery
  - Application version is "Baselined"
- Test and Production Deployment
  - Release package is "In Test"
  - Release package is "Ready for Production"
  - Release package is "Deployed to Production"

The following chapter provides more insights on the quality gates and what requirements must be met to pass them.

Backlog items

Apps

Release packages

Backlog

Implement → App Delivery → Test and Production Deployment

Backlog item is Ready

Application is Baselined

Release package is In Test

Backlog item is Done

Release package is Ready for Production

Release package is Deployed to Production
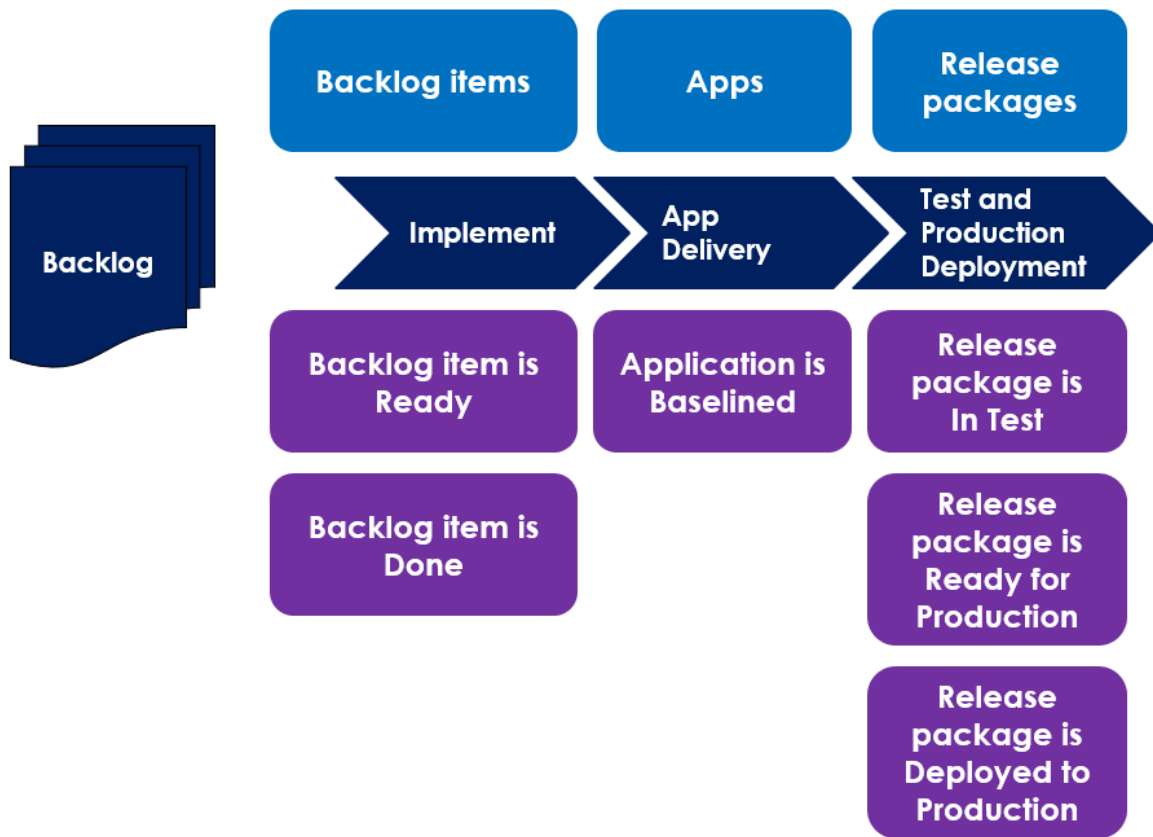
Process Overview with phases, instances, and quality gates

Quality Gates

Implementation: Backlog Item is "Ready"

To pass this quality gate a backlog item must be sufficiently documented, so that the development team can start working on it.

Sufficient documentation must meet all criteria defined in the "Definition of Ready". However, not only must the "user story" be told, but there are also architectural decisions to be made.

Which applications need to be created or changed? Which dependencies must be considered? Should existing functionality be changed or new components be created?

Another important aspect is the relevant personas, which ideally map to roles and access rights.

As always a fair balance needs to be found between excessive precision upfront and flexibility and creativity in the process. Asking these questions is never wrong – but in some cases they may get answered later in the process.

The question whether any kind of estimate (in hours, story points, or t-shirt sizes) should be done before or after a backlog item is considered "Ready". One can argue that an estimate is only possible IF the backlog item is ready. One may also argue that once enough information is available for an estimate, it could also be made a requirement to include the result of an estimate into the definition of ready right away. As many organizations develop their own agile style, this document does not make any suggestion when estimation takes place or even if estimations should be done at all.

Checklist

- The documentation of the backlog item is compliant with the Definition of Ready
- The impacted applications have been identified and documented
- The expected dependencies have been identified and documented

Relevant Building Blocks

- Application Architecture and Dependency Management
- Definition of Ready

Gatekeepers

- Lead Developers
- Architects

Implementation: Backlog Item is "Done"

To pass this quality gate a backlog item must be completely implemented according to the requirements documented in the backlog item itself AND the additional requirements documented in the "Definition of Done" – which should specify that all tests must pass, and the delivered code is fully compliant with the Coding Guideline.

The completed code must be committed to the Source Code Repository.

"Done" does not mean that the capability described in the backlog item is available to users in production.

Checklist

- All activities defined in the Definition of Done are completed
- All related changes have been committed to the development branches of the impacted applications
- All related application files are compliant to the coding guideline

Relevant Building Blocks

- Definition of Done
- Coding Guideline
- Automated Tests
- Source Code Repository

Gatekeepers

- Lead Developers
- Product Owners

Application Delivery: Application Version is "Baselined"

To pass this quality gate an application version is available as a version specific branch (or tag) in the source code repository.

All activities described in the Application Version Baseline Procedure must be completed. That includes that any unfinished work is configured in a way that it does not impact the application's behavior and it is (like all other technical debt) is documented as outlined in Technical Debt Management. The development team must have run all automated tests and verified their compliance with the Coding Guideline – verified through Instance Scan checks.

Checklist

- All activities defined in the application version baseline procedure have been performed
- The version branch is locked and protected against future changes

Relevant Building Blocks

- Application Version Baseline Procedure
- Technical Debt Management
- Source Code Repository
- Branching Strategy
- Coding Guideline
- Automated Tests

Gatekeepers

- Lead Developers
- Architects
- Product Owners

Test and Production Deployment: Release package is "In Test"

To pass this quality gate a release package is successfully installed on a TEST environment.

To get there, a set of application versions – which typically relate to each other through their documented dependencies – must be identified. All steps required to install these application versions must be documented, reviewed, and performed on the TEST environment. As part of the deployment, all automated quality assurance activities should be conducted – ideally even before installing the release package on the TEST environment. These automated activities should at least include running all available and relevant ATF tests contained in the applications in the release package and performing coding guideline compliance checks using Instance Scan checks on these applications.

Checklist

- Deployment instructions do NOT contain update sets (all changes must be shipped as part of a baselined application version)
- Deployment instructions do NOT contain manual tasks – everything that needs to be "done" should be scripted in installation scripts
- All applications contained in the release package have been deployed to the TEST environment (in the required versions)
- All applications are fully compliant with the coding guideline – i.e., there are no Instance Scan findings
- All installation scripts of all applications have been executed
- All ATF tests contained in all applications have passed
- All steps defined in the deployment instructions have been performed – this may include activities to be performed in external systems or communication to stakeholders
- Smoke tests have passed – these are any quick manual tests

Relevant Building Blocks

- Source Code Repository
- Coding Guideline
- Automated Tests
- Deployment Automation

Gatekeepers
- Test Managers
- Operations

Test and Production Deployment: Release package is "Ready for Production"

To pass this quality gate a release package is successfully tested on a TEST environment.

This document does not make any statement regarding manual testing – be it scripted or explorative – nor does it contain guidance on automated testing performed by tools outside ServiceNow. Such activities for sure play a significant role to verify quality and should be conducted with great care and craftsmanship.

If any of these tests fail, however, defects must be reported to the development team. The Defect Management and Backporting procedures specify in detail what – besides fixing the defect – needs to be done and where.

The most important aspect in this quality gate is that only if all tests pass, only then the release package in the tested form can be deployed to production and considered as "Ready for Production".

If there is at least one show-stopping fail, the responsible applications must be fixed and provided as new versions. This constitutes a new release package, and the tests must start over.

Checklist

- All tests have passed OR all failed tests are not considered showstoppers
- All identified defects have been documented according to Defect Management and Backporting

Relevant Building Blocks

- Defect Management and Backporting

Gatekeepers

- Test Managers
- Operations

Test and Production Deployment: Release package is "Deployed to Production"

To pass this quality gate a release package must be installed on the production instance. This may include to make the newly introduced capability available to users, but it does not have to. In some cases, new capabilities are introduced in a silent go-live without visible impact on users.

Checklist

- All applications contained in the release package have been deployed to the production environment (in the required versions)
- All installation scripts of all applications have been executed
- All steps defined in the deployment instructions have been performed
- Smoke tests have passed

Relevant Building Blocks

- Source Code Repository
- Deployment Automation

Gatekeepers

- Operations
- Product Owners

Process Roles

The following roles are required to execute the proposed process. Multiple roles may be assigned to a single person and some roles may be assigned to multiple individuals – depending on organization structure, culture, and team size.

Platform Owner

The platform owner is responsible for the platform and its operations. The platform owner should have decision making power with regards to staffing and budgets.

The platform owner is the highest escalation instance for any conflicts that cannot be resolved by other team members.

Developers

Developers convert requirements documented in backlog items into new and changed applications.

Lead Developers

Lead Developers are responsible to baseline applications and to oversee the compliance to the Coding Guideline.

Architects

Architects research, advise and decide on solution approaches. They are responsible for the application architecture – they own the optimization problem to balance the amount, and size of applications and the reduction of dependencies between applications.

Deployers

Deployers conduct deployments of release packages to downstream instances. They own the process of the deployment and its automation.

Operations

Operations are responsible for the day-to-day administration of the platform. In case of production incidents, they are responsible to provide solutions. They are responsible to monitor the production platform and can respond to monitoring events. They perform configuration changes that are independent from deployments.

Testers

Testers confirm compliance of release packages to set expectations.

Governance Stakeholders

Governance stakeholders are responsible to define checks and procedures to be applied at the quality gates – and to provide insights into process quality from their area of expertise. They may also add backlog items to support this. Every governance stakeholder has a say in backlog item prioritization.

Refer to chapter "Governance" for details about the various governance stakeholder roles.

Governance

A ServiceNow platform requires an effective governance body. Of course, every organization has its own ways to structure, staff, and organize its governance body. This chapter outlines activities and responsibilities of the constituents of the governance body. Eight stakeholder "roles" are required to oversee the operations of a ServiceNow platform. How and by whom these roles are taken is for the organization to define.

Each of these stakeholders can add backlog items to the backlog and should have an equal say on how to prioritize these backlog items.

Each stakeholder can also define criteria by which the gatekeepers should consider and verify when making decisions and letting "process instances" pass their quality gate.
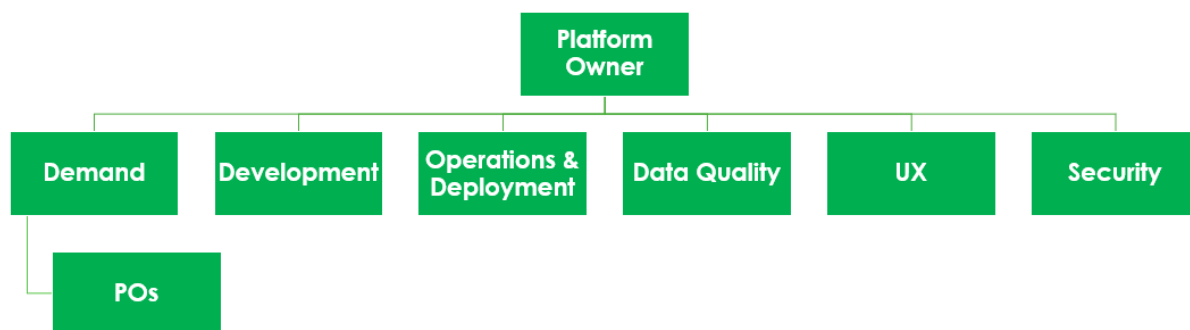
Platform Owner

As the name suggests the platform owner "owns" the platform. The platform owner has ultimate decision power and is the highest escalation instance for any conflict or decision to be made.

In the context of the governance body, the Platform Owner is responsible that each stakeholder role is staffed.

Demand

The "Demand" stakeholder is responsible for the identification and the prioritization of requirements and the approval and testing of new and changed capabilities of the platform. Usually the day-to-day work of identifying, prioritizing and documentation of requirements is delegated to one or more "Product Owners".

Product Owners

"Product Owners" are responsible to identify, document and prioritize requirements related to a defined area of the platform or one or more business or IT processes.

Development

The Development stakeholder is responsible for the output quality of the work of all developers. They own the Coding Guideline and those automated tests that are created and maintained by the development team.

Typically, the Development stakeholder demands criteria for the documentation of requirements before they are being worked on by the development team. These criteria or documented in the Definition of Ready.

Operations and Deployment

The Operations and Deployment stakeholder is accountable for all activities that administrators perform on all instances, and the deployment of applications to non-development instances.

The main concern of the stakeholder is operational stability. A key element is that what is delivered by the development team is deployed to target instances exactly as intended. Only then can testers verify shipped release packages and confirm that they are fit for purpose and that these can be deployed to production.

Data Quality

The "Data Quality" stakeholder is accountable for correct and complete data on the production instance. They also oversee that any data that should no longer be present on the production instance is archived and removed.

Each table of the ServiceNow that contains data (as defined in chapter "Data vs. Code") should be associated to a stakeholder. It is unlikely that one person can manage that responsibility alone – so this stakeholder role should be shared between multiple stakeholders depending on the data.

Data Quality stakeholders may demand new or updated Instance Scan checks to be developed so that data quality can be monitored and managed on the production instance.

User Experience

The "User Experience" stakeholder's main responsibility is to ensure that features are relevant, useable, and accessible to users. This stakeholder may have a say in the way new requirements are identified and described and how the user interfaces for these new capabilities are designed.

Once made available to users these capabilities can be checked through various means – from user interviews to statistical analysis of usage behavior. The "User Experience" stakeholder oversees these activities.

User satisfaction is their main concern – as adoption is driven by user experience and the feeling that an application is providing value.

Security

The "Security" stakeholder oversees all activities targeted to improve security on the platform. This starts with proper documentation of requirements considering what specific persona can or must not be able to do or access. It involves automated and manual testing activities and corresponding monitoring in production.

Testing

Testing is a fundamental part of the process. Every single quality gate, building block, and every technique has some kind of testing built into it. This chapter provides an overview on the terminology and the different intentions (why we test), scopes (what we test), methods (how we test), and tools used for testing.

Testing is so important that it is covered by a separate chapter – it is a binding element across all aspects of the process.

For a team to communicate effectively about testing, the terminology must be aligned. This chapter aims at providing a common language about testing.

Doing the right thing and doing things right

Validation is to verify that we are doing the right thing.
Verification is to verify we do things right.

These are different aspects of testing, they require different approaches, different techniques and take place at different times.

Validation is about creativity, empathy, understanding, anticipation, investigation – it is part of a design process that never ends.

Verification is about precision, details, the happy path and the potential errors, the thinkable edge cases, the nitty gritty stuff, looking under every stone.

Verification is – for the good or the bad – agnostic to whether what has been built is easy-to-use or makes sense from a business perspective. Verification is about making sure things work as described and designed.

It is validation that helps a team to understand whether the intended outcome is likely to be achieved, whether users will understand, adopt, and make the best use of provided technical capabilities.

This distinction has implications on the approach, tools, and techniques – and eventually on the personnel that conducts the corresponding tests.

In an ideal world, verification is done BEFORE a backlog item is considered ready. Then AGAIN during the formal test conducted on a test environment. And continuously AFTER applications are deployed to production and exposed to users.

Validation on the other hand starts with the implementation and is performed continuously throughout the delivery process – and – in the form of operations monitoring – continued after go-live.

Intentions – why are we testing?

- Validation of designs and requirements
- Validation of the intended outcome when exposed to users
- Verification of functional requirements
    - Narrow scope technical capabilities ("units")
    - Use cases (Sometimes expressed as user stories)
    - Full end-2-end processes
    - "Business Capabilities" that may contain multiple processes, integrations, etc.
- Verification of non-functional requirements
    - Performance
    - Security
    - Maintainability
    - Portability
    - Scalability
    - Coding Guideline compliance
    - Documentation
- Protection from Regressions
- Verification of defect resolution

Scopes – what are we testing?

- Unit
  (the smallest possible technical element)
- Capability
  (A combination of technical units that provide some distinct value to users or stakeholders)
- System / Integration
  (A combination of capabilities and external systems – real or simulated)

Methods – how are we testing?

- Manual
  - Ad-hoc / Exploratory
    (Testers interact with the system without a clear script or path to follow – in some cases testers may be given a goal but no guidance on how to achieve the goal)
  - Scripted
    (Testers interact with the system following a clearly defined path and given a clear set of quality criteria on when the test is to be considered failed or successful)
  - Code review
    (Peer developers look at code and join a discussion about what they see and understand)
- Automated
  - Whitebox
    ("Whitebox" means that the test tool has both access to the defined interface AND to the underlying structures, e.g., the database tables being modified during the test – so the result of the test may not only depend on the output provided by defined interfaces but also by assessing the actual changes to the database or external systems)
    - Simulation
      (Test scripts execute specific parts of the system and compare actual and expected results)
    - Static code analysis
      (Test scripts are used to analyze the source code artifacts directly without executing any of the analyzed code)
  - Blackbox
    ("Blackbox" means that the test tool has ONLY access to the system via defined interfaces – and the test result is assessed based on the responses provided by these defined interfaces)
    - Via API
    - Via GUI
    - Via data import/export interfaces (e.g., SFTP, file shares, database connections, etc.)

Tools – what do we use for testing?

- Automated Test Framework
  The ATF is the ServiceNow test automation tool that enables automated scripted white box and black box testing and to ship the tests as part of applications.
- Instance Scan
  Instance Scan is a ServiceNow platform capability that can be used to perform white box testing on code and configuration artifacts at rest – that is to check the source code and the configuration instead of executing it.
- Test Management 2.0
  A ServiceNow platform capability to manage and document manual scripted tests.
- External tools (E.g., Selenium, RoboClient etc.)
  Any other tools for automated or manual black box or white box testing or the documentation of such tests.

Test Suites that yield value over time

Testing is effort. Yet, a well-balanced and executed test strategy can reduce overall efforts, costs and increase trust of all stakeholders over time.

A **Test Suite** is the superset of repeatable tests being performed as part of the process.

This does not include any ad-hoc or exploratory tests.

While all tests that are conducted manually directly translate into effort each time the tests are executed, the long-term effects on efforts of automated tests are more subtle.

Obviously the initial creation of automated tests is associated to some effort – which if done very early in the process (during or even before actual development) is not only hard to measure but – that's the good news – almost insignificantly low.

The tricky part is the maintenance of a Test Suite over time. How much effort is required to adapt an existing Test Suite to modified components? That is the key question.

The better a Test Suite is designed – the less costs it will require over time – and the more value it will yield – by detecting introduced regressions early in the process – when their remediation is least expensive.

What can be said about the long-term costs of a code base can also be applied to a Test Suite.

The more code there is, the higher the maintenance cost. If less code (in a wider sense) is required to implement the required capabilities, more value is produced.

The exact same is true for tests in a Test Suite.

When talking about code, the most obvious pattern to look after is duplication. The more often the exact same code pattern, logic or sequences repeat themselves in a code base, the more code is needed. Duplication, however, may come in very subtle forms. Exact copies of functions or even classes may be spotted here and there, but most cases of duplication are more subtle and often not that easy to detect – even for the experienced eye.

It requires intimate knowledge and oversight over large parts of a code base to spot the more subtle repeating patterns for the human eye – and so far even very advanced static code analysis tools were not convincingly good at that.

Rather it requires a mindset shared by the whole team to always seek for logic that can be refactored into re-usable components. This will shift the task from spotting duplications in hind-sight towards making code as re-usable as possible all the time – which will eventually lead to code being moved away from a very specific context into places where the team agrees to put re-useable components – where it is then easier to see if a comparable components already exists and whether – maybe with slight modifications – it can be used for the specific case a developer currently works on.

This same principle should be applied to tests.

When done so, the resulting Test Suite is likely to consist of many smaller, narrow scoped tests, and fewer larger, broad-scoped tests. It will consist of tests that verify the functionality of single components, rather than many components at the same time. Tests that do not depend on each other, data, the environment, the time of day.

If most tests are of that nature – and that applies to automated and manual scripted tests – the resulting Test Suite will be easier to change and adapt and hence to maintain.

Test Data

To perform tests in many cases test data is required. To test different scenarios and use cases different combinations of data is needed during the execution of different tests. And some of the data combinations may not exists at the same time.

So, the creation and maintenance of test data on different instances to support different test scenarios and to different testing methods, intentions and scopes is all but trivial.

Test data is opposed to production data is any data (not code) that is used to conduct tests. In rare cases, test data may even exist on a production instance and that data may be stored in the same tables as production data.

With that being stated, it should be obvious that test data must be created, managed, maintained, and eventually be removed consciously and according to an agreed and defined process. Ownership should be defined for test data on different instances.

Consider the following guidance on test data:

- Test data must be obvious – both the human eye and software should be able to tell if a given record is test data or not
- Automated Tests must not depend on any existing test data on an instance: all data required by the automated test must be created by the automated test
- Test data should be generated through code – not manually
- Test data ownership must be clearly defined per instance and table

Test Users

Test users are test data artifacts. However, they are special. They are not only part of a test data constellation that is required to test specific scenarios or uses cases, but they are used to perform the tests – as test users are impersonated by manual testers or automated tests.

Test users should hence be as close to real users (human or technical) as possible to produce high fidelity test results.

As outlined in chapter "Personas and Roles" users should always be associated to a persona role – the same can be said about test users for end-to-end tests – but in some cases only specific technical roles should be assigned to test users to verify the exact behavior and access defined for specific roles.

The platform does not have a specific representation of test data – including test users in its OOTB data model. There is no flag by which test data and test users could be identified.

This calls for a naming convention to identify test users.

The following convention worked well for several teams for several reasons:

- A test user's "First name" is ALWAYS "Test"
- A test user's "User ID" ALWAYS starts with "test."
- A test user's "Last Name" contains a hint on the persona or role which the test user has (e.g., "Service Desk Agent", "Report Admin", or "ITIL")
  If applicable the application name that defines the role should also be used in the "Last Name" (e.g., "Agile Agilist", "Deployer Manager")
- A test user's "User ID" ALWAYS ends with the lowercase, dot-separated equivalent of its "Last Name" (e.g., "test.servicedeskagent", test.report.admin", "test.itil", "test.agile.agilist", "test.deployer.manager")
  Note how the dot is used to express a hierarchy in the naming scheme.

By following that convention, developers and testers can

- Filter for test users
- Identify a test user as the source of a log entry or any kind of errors

The Definition of Ready ideally defines some test cases which should be created as part of the backlog item. All test cases require a test user; hence the description of test cases should specify which personas (or roles) should be used to perform the corresponding test cases.

Building Blocks

Application Architecture and Dependency Management

In the past, individual changes (or "customizations") were done and captured in update sets. These update sets were then deployed selectively. And that was ok, if there are only a few dependencies and their sequence either did not change or did not matter.

Some clients have built their own methods, processes, and tools for handling the "update set hell" in more complex scenarios. Kudos for that.

This document proposes a fundamentally different approach: Everything is shipped as an application. That can be

- a scoped application (preferred)
- a global application (should be the exception)
- a customization application (if there is no other alternative to change a plugin's or store app's behavior)

Scoped applications are always the first choice. From an architectural point of view, these applications are the modules that constitute the architecture.

Architects have several decisions to make, and to make this process work, the most crucial decision is how to split the business capabilities into technical capabilities and how to slice and dice these technical capabilities into one or more applications.

That is the primary task for architects. The business describes a demand, the development team and the architects draft a solution, and decide:

- which applications should be changed
- what new applications should be created

The superset of these decisions is what we call the "Application Architecture". Ideally there is an up-to-date visual representation and documentation of that architecture for reference and for new developers to understand what is going on.

When we talk about applications as modules, some of these modules might relate to each other. More precisely, they depend on each other.

This is where "Dependency Management" comes into play.

As important as the decision on which technical capability should be implemented into which application, so is to keep track about the dependencies between these applications.

First things first: What are dependencies?

In general terms application "A" is dependent on application "B" if

- "A" calls a function in "B"
- "A" extends something in "B"
- "A" just only works if "B" is also installed

In ServiceNow terms, "A" is dependent on "B" if (for example)

- "A" uses a Script Include contained in "B"
- "A" extends a table defined in "B"
- "A" adds a field to a table defined in "B"
- "A" reads or writes records from or to a table that is defined in "B"
- "A" requires a business rule, flow or other logic contained in "B" to work properly

After establishing the existence of a dependency between two applications, we also need to consider their versions. The dependency of one application on another may change over time and not all versions depend on each other in the same way.

The statement: Application "A" depends on application "B" is not sufficient – in fact it is a specific version of application "A" that depends on one or more versions of application "B".

Imagine a case where application "A" version 1 depends on a function that has been introduced in application "B" version 2.

Application "A" version 1 depends on application "B" version 2 (or later) – you may refer to chapter "Sematic Versioning" for more details on versions, their compatibility and meaning.

To know the dependencies between all applications serves multiple purposes:

- We can derive the exact order in which these applications must be installed during deployment
- We can determine which other applications must be installed (or even baselined) before an application can be deployed
- We can assess the impact of potential backward incompatible changes of an application on its consumers (the applications that depend on the to be changed application)

One can think of the superset of all applications in an instance as a forest of one or more dependency trees. Some applications are roots as no other applications depend on them. But these root applications may depend on other applications and these applications may then depend on even other applications – forming branches and leaves of the dependency trees.

To simplify this, one can transform the forest into a single tree.

Enter the **"Platform" scoped application**. Each instance should have its own platform scoped application that serves as the one and only tree root for all other applications. The version of the platform application could be interpreted as the version of an overarching release. The dependencies of the platform application define which versions of all other applications must be deployed as a release package.

The Platform application really is a scoped application with its own repository in the Source Control System – which needs to be maintained by the development team and baselined whenever a new release is prepared.

Once Deployment Automation has reached the highest grade, a deployment can be as simple as "just" installing the platform app (which will then lead to the installation of all its dependencies in the required versions).

The metaphor of the tree points to another detail which is important: Dependencies must ALWAYS be unidirectional.

If application A depends on application B, application B MUST NOT depend on application A.

This is also true for indirect dependencies: If application A depends on application B and application B depends on application C, application C MUST NOT depend on application A.

Such a scenario would form a circular dependency which is forbidden.

An application and its dependencies can be represented as a **dependency tree** where the root is the application on top and its dependencies are the branches, and their dependencies are the smaller branches, and their dependencies are the leaves and so on. Some applications may appear multiple times in the tree.

An example:

- App Alpha
  - App Beta
    - App Epsilon
    - App Gamma
      - App Zeta
  - App Gamma
    - App Zeta
  - App Delta
    - App Eta
      - App Theta

The example should be read as "App Alpha" depends on "App Beta". "App Beta" depends on "App Epsilon" etc.

Also note that in the real world each of the dependencies must also include a version requirement.

This dependency tree can also be represented as a **dependency list** which indicates the order in which application must be installed:

- App Theta
- App Eta
- App Delta
- App Zeta
- App Gamma
- App Epsilon
- App Beta
- App Alpha

Each dependency tree may be transformed into multiple valid dependency lists. In this example, the following list is also valid:

- App Zeta
- App Gamma
- App Epsilon
- App Beta
- App Theta
- App Eta
- App Delta
- App Alpha

Note that "App Gamma" appears at different positions in the two variants of the dependency list, but "App Zeta" is always installed BEFORE "App Gamma" because "App Gamma" depends on "App Zeta".

In any case an application can only be installed after all its direct and indirect dependencies are installed in the required versions.

This rule must be strictly followed. The most important reason is this: Given an application Alpha adds a field to a table that is defined in application Beta, application Alpha depends on application Beta. If application Alpha were installed before application Beta is installed, a new table would be created in Alpha's scope containing only the added field. A subsequent installation of application Beta would fail and only database administrators can fix the issue.

To perform Dependency Management on a day-to-day basis, the platform must be extended with the following capabilities:

- Document dependencies from one application to another allowing multiple (major) version requirements
  - Such dependencies must be part of an application so that they are also stored in the application's source code repository
  - The dependencies must be stored in a way that they can be retrieved and processed from a source code repository
  - The dependency tree must be visible to developers
  - The dependency list shows all required dependencies in the exact order in which they must be installed
  - Any issues (like circular dependencies) must be detected automatically

These requirements are not met by the OOTB dependency table provided in ServiceNow – hence this additional capability must be implemented.

The open-source application "DevTools" supports these capabilities.

As in most cases, by far the most relevant factor is people. Without a team of architects – or developers who perform that role and take the responsibility – any technical tool is pointless.

Setting up a team of people who really care for the application architecture, drive its development, create a vision, and pursue a long-term strategy is the key success factor.

Definition of Ready

The Definition of Ready is the central building block for the first quality gate. It is important that the team has a say in what should be contained in the Definition of Ready. However, it is strongly recommended to include the following points:

- Description of desired outcome / Use case
- Affected applications
- Affected personas and technical users
- Affected process(es)
- Affected data entities
- New and modified fields (including labels)
- Affected GUIs (Classic, Portal, Mobile)
- Relevant GUI languages
- Affected notifications
- Affected technical interfaces
- Required access restrictions
- Data Protection/GDPR/Compliance implications
- Required documentation
- Proposed test data
- Proposed test cases

When presented to teams for the first time this list may cause some concerns – especially amongst product owners or other stakeholders representing the business. They might argue that a good portion of these topics cannot be provided by them.

But passing the first quality gate is NOT a hand-over from product owners to developers!

It is important to understand that it remains a team effort to describe a backlog item in a way that it is compliant with the Definition of Ready. It requires the business to describe the need, the architects to make decisions on which applications to change and the development team to fill in implementation details where needed.

Definition of Done

Obviously the "Definition of Done" is the central guidance and checklist for the second quality gate: Backlog item is "Done".

In many agile organizations, both the "ready" and the "done" state are seen as hand-over points where the work (or responsibility for something) is handed over from one team to another. This perspective might seem comforting, but it is not helpful.

This process will not work well when things are thrown over the fence, and then thrown back if they do not comply with a set of rigid rules.

At all stages it remains a team effort – to define the rules, to use them, to apply them, to review and change them.

The definition of done is often used for protection by a development team – where a team tries to defend a way of working which they perceive as good quality against business stakeholders who want to get new features out of the door as fast as possible at the lowest possible cost.

Deployment Automation and the embedded use of automated quality checks is intended to bring some relief to this conflict. Once there is a baseline of quality requirements which are checked as part of every deployment, both the developer team and product owners know that if they do not deliver according to the agreed quality levels – nothing will be shipped. Period.

The Definition of Done is a pre–check after the completion of a single backlog item – before the feature is shipped as part of an application version. The line items in the following proposal should hence be considered as non-negotiable, others can be added at the team's discretion. The question is less "if" these checks should be performed, the question is rather "when" they should be performed:

- All described features are implemented
- Implementation is peer reviewed
- All described (and relevant) test cases covered in ATF tests
- All related (new or changed)ATF tests are peer reviewed
- Compliance to the Coding Guideline (i.e., no Instance Scan findings)
- All related ATF tests pass
- All technical documentation written
- All changes have been documented in the Release Notes
- Validated by Product Owner (explorative test)
- Application dependencies are verified and updated if needed

Coding Guideline

It may appear self-evident to make a coding guideline part of a mature development and deployment process. It is not. Only a few teams or organizations have coding guidelines, and even fewer actively use them. But why is this the case?

Good coding practices are expected to lead to better code, which is easier to understand, easier to maintain, easier to analyze, easier to test, easier to change. Consistency across a code base produces value by itself, since developers can then navigate the code base better and are able to make changes to different parts of the code – those parts they were not actively working on themselves.

A coding guideline may not only describe how specific code details should look like, but it can also describe which automated tests should exist and how code should be structured – which types of functionalities should be implemented where and thus may even go beyond being a guideline for code – but even be a guideline for the architecture.

ServiceNow provides many low-code and no-code capabilities. The outcomes produced with these tools should also be covered in the Coding Guideline. The "Coding" in Coding Guideline should be interpreted as broad as possible.

A Coding Guideline may cover:

- The use or non-use of specific coding patterns (or anti-patterns)
- Error handling
- Naming conventions
- In-code documentation
- Test coverage
- Modularization and how to split things
- …many other aspects

A coding guideline is a set of rules – to which a team has agreed. That points to a very important aspect: Participation. If the Coding Guideline is a law imposed by a privileged small group to control the work of a larger developer population, we should not expect any positive impact.

The coding guideline should be the result of a collective experience of failures and mistakes. The reason why a team uses Coding Guidelines is to avoid making the same mistakes and experiencing the same failures again. The Coding Guideline should be an expression of collective learning.

To make this happen the whole team must have the opportunity to participate and contribute, to have a say in which rules should be applied, and which are not and which exceptions to the rule should be allowed in which cases. For sure, consensus cannot be achieved in all cases – so some form of a decision-making process is needed. Please refer to chapter "Communities of Practice" for guidance.

Once a team has agreed on a coding guideline (and on a process to change it going forward) it needs to be applied. Just writing down what the team intends to do is not good enough. To be straight: the coding guideline document alone will not have any significant impact on the code base. The individual rules described in the guideline must be checked – automatically – by the developer when writing code and as part of every single deployment.

The OOTB platform feature "Instance Scan" provides the framework to build checks that can investigate every single record (of an application) and assess whether it is compliant to the Coding Guideline. Further, the checks and their documentation can be used to generate the

Coding Guideline document – making the checks the actual single source of truth of the Coding Guideline. Instance Scan essentially is turned into a static code analysis tool within the ServiceNow platform.

With that being said, the recommendation is to limit the Coding Guideline to statements that can be checked automatically – and hence produce a binary output of either compliance or non-compliance with no room for interpretation.

This is important for two reasons: running an Instance Scan check as part of the deployment pipeline requires a clear decision on whether the deployment should proceed (no findings produced by any check) or not (at least one finding was produced by a check). A developer should get clear guidance on whether the code they wrote is compliant or not – any room for interpretation will lead to ignorance in the long run.

If, however, a check reveals a finding and the developer is yet convinced everything is fine and as it should be, there should always be the following options:

- The intention of the check is valid and agreed by the team, but the implementation of the check does not match that intention, the check must be improved
- The team does not agree on the intention and decides to remove the check
- The team confirms the intention and the correctness of the result, and the developer get all the support needed to refactor the code and the time and support needed to understand the intention and solution

This feedback loop is important for organizational learning and to build awareness and support in the team.

A team does not need to start from scratch with a blank page, there are numerous resources available that can serve as a basis for the team's Coding Guideline. There are several publications on ServiceNow good practices (not to say "best practices"), numerous publications on JavaScript coding practices in general and even several open-source contributions for check implementations to start with.

However, to integrate the Coding Guideline related Instance Scan checks into the developer workflow and into the automated deployment the following technical capabilities are required (which are not available OOTB):

- Option to run an Instance Scan check on an application that only uses checks related to the Coding Guideline
- Option to run an Instance Scan check on a single application file that only uses checks relevant for the Coding Guideline
- Creation of the Coding Guideline document based on the relevant checks (to be published in a knowledge base, a corporate wiki, or any other knowledge management system)
- Mechanism to ignore specific checks on specific applications (blacklisting checks)
- Mechanism to use specific check only on specific applications (whitelisting checks)

The open-source tool "CodeSanity" has all these technical capabilities and contains numerous implemented checks which embody the learnings of several large teams working on highly complex application eco-systems.

When a new set of checks are applied to an existing code base, the new rules will inevitably clash with the existing code. The team needs to figure out what fits, what they need (and like), where exceptions need to be made for the existing code and where investments should be made to refactor code.

Automated Tests

The key to shorter release cycles are automated tests.
The key to fewer regressions are automated tests.
The key to fewer defects are – again - automated tests.

Automated tests are crucial for a robust, reliable, and mature software development and deployment process.

Titus Winters, one of the authors of *Software Engineering at Google* stated: "One of the broad truths we've seen to be true is the idea that finding problems earlier in the developer workflow usually reduces costs."

This underlines the idea – often referred to as "shift left" – to establish quality as early in the process as possible.

There are different forms of automated tests, and such tests may be executed for different reasons and different stages of the process.

Of the many forms of automated tests (and tools) which may be helpful to verify quality in a specific organization, the automated tests that can be developed and deployed as part of the application and run within the platform are most relevant. In ServiceNow these are tests based on the Automated Testing Framework ("ATF"). The ATF is an OOTB platform feature.

ATF – or to be precise – the many ServiceNow plugins and products come with a set of ready-built tests. These quick-start tests validate the OOTB features in their unaltered non-customized form. However, most relevant for this development and deployment process are the tests that the development and test team create to verify the result of their work.

The recommendation in this document is to

- Practice Test Driven Development – and create tests as early as possible by the development team (refer to chapter "Test Driven Development" for details)
- Create tests as part of the applications to be shipped
- Execute tests as part of every deployment
- Maintain zero-tolerance regarding failed tests

Executing tests is an integral part of the automated deployment and only if all tests pass an application be deployed to a downstream instance. That requires discipline regarding the fidelity of all tests contained in an application regarding its behavior. A zero-tolerance mindset can only be applied if tests are kept up to date with the application on the long run.

To be executed continuously and successfully as part of the deployment and to be maintainable on the long run, tests should meet the following requirements:

- Tests are self-contained – that means they are agnostic regarding the data that already exists on the instance. Required test data must be created by the test – that includes users and groups and any other record that is relevant for the outcome. This is very important yet sometimes difficult to achieve.
- Tests must be executable in headless mode. This may be achieved using the Cloud Runner – for tests that make use of client-side test steps or by using server-side test steps only.
- Tests should not be redundant – every test should verify a specific aspect of the application's behavior; any duplication should be avoided. The scope of a single test

should be as narrow as possible. Lengthy end-to-end tests are more difficult and costly to maintain.

- Tests are part of an application and verify the behavior of that application – by that these tests indirectly verify the behavior of the application's dependencies. However, the results of the tests must not depend on the behavior of applications that depend on the application being tested. As a result, tests contribute to the dependencies of their application and like with any other application component bi-directional (i.e. circular) dependencies must be avoided.
- All tests that are intended to be executed as part of the automated deployment should be contained in a test suite that is named exactly after the application – so that the tests can be clearly identified – and of course be contained in that application.
  E.g. an application named "My App" should contain a test suite called "My App". This test suite is to be called during deployment.
  The application may have any number of additional test suites (e.g. for GUI based tests that cannot be run during deployment – the recommended name for the test suite containing GUI-based tests is "My App GUI")

"Tests are not outside the system; rather, they are parts of the system that must be well designed if they are to provide the desired benefits of stability and regression." *Robert C. Martin, Clean Architecture*.

Writing automated tests has an impact on how software must be designed. Teams that start writing tests late in the process will face all sorts of challenges. Obviously this insight is of little help when a development team is faced with already written code and applications with little test coverage.

However, that must not be accepted as an argument against making the first step and starting the journey.

In reality the increase of test coverage has always a positive impact on code structure, architecture, and overall maintainability of code – as there is a natural incentive to refactor (existing) code to allow better testability.

Sometimes stakeholders may oppose the idea of automated testing as there is a widespread – and false – belief that maintaining a suite of automated tests implies higher costs and prolonged time-to-market cycles. While this observation may be true for the first iterations after a development team started the new practice even short to mid-term effects are the opposite. A well-crafted test suite increases velocity, reduces roundtrips between development and test, reduces production issues and results in better maintained code.

Never ever argue about the costs of automated testing. Never adjust estimates by skipping automated tests.

Defend that ground at all costs!

Source Control System

No matter which variant of the process you choose, which level of automation, how broad the test coverage or how detailed your Coding Guideline is – in any case you will need a Source Control System.

This can be

- Private or even public repositories on GitHub
- A self-hosted or private instance of GitLab
- Any other source control system supported by ServiceNow

Many organizations have requirements to store code and/or all deployed application versions in their self-operated Source Control System (e.g., GitLab) or definitive software library (e.g., Jenkins).

AppRepo – the ServiceNow hosted "Application Repository" – however does NOT qualify as a Source Control System. AppRepo can deploy published – that is "baselined" applications to downstream instances. Think of AppRepo as the ServiceNow equivalent of a definitive software library (in ITIL terms) (e.g., Jenkins) – not more, not less.

To make this process work, we will need both: a system to safely store ServiceNow applications while they are being worked on and to reliable deploy these applications to downstream instances.

Whether a Source Control System should be combined with the ServiceNow hosted AppRepo is debatable.

ServiceNow official documentation – at the time of the publication of this document – advises against the use of a Source Control System to perform deployments to production instances. The main argument is that deployments via source control may lead to the deletion of records and data – if such records (including table and column definitions) are (no longer) part of the application stored in the source control branch being deployed. This however is exactly the expected and intended behavior and perfectly aligned with the process proposed in this document.

The customer environments where the insights were gathered that led to this document all did not include the use of AppRepo.

Here are some reasons that resulted into the decision not to use AppRepo:

- On-premise hosting and air-gapped – and hence no access to ServiceNow AppRepo
- Limitations regarding the amount of stored application versions in AppRepo – this deficiency is reported to be fixed in the meantime
- Limitations regarding the total size of application file data to be stored in AppRepo
- Additional risk and maintenance work for pipeline automation using two different deployment methods (via source control system AND AppRepo)
- Inability to publish hotfix versions with a lower version number than the latest published version – this deficiency is reported to be fixed in the meantime

For these reasons, the guidance in this document is to use a Source Control System both for development and for deployments – including deployments to production.

The process and all technical and operational details are agnostic to the use of AppRepo. To leverage the benefits of this process its use is not required.

Branching Strategy

Most experienced developers have at time point in their career worked with a source control system and many of them have been exposed to widely adopted standards like gitflow – which essentially proposes the use of feature branches.

The main idea of feature branches is that any separatable feature is developed in its own branch and once the work on that feature is completed, that branch is merged back into a master (or default) branch.

This approach works under the assumption that source code can be merged by the Source Control System. Merging involves a side-by-side comparison of source code files and the possibility to make decisions about how to inject the proposed changes taken from the feature branch should be ingested back into the master branch.

That works fine with traditional source code files, config files, even xml files to a certain extend.

And here is the problem: branch merging does NOT work with all ServiceNow application files.

Here is why: Traditional source code is text only.

ServiceNow application files are stored as records in the database and once they are committed to the Source Control System, these records are represented as xml files – which do not only contain source code but also meta data and – this is the tricky part – references to other application files.

For some application files, like Script Includes and Business Rules – which consist of exactly one record -, the process of merging – even when represented as xml files – is working.

ServiceNow however provides a growing toolset of low-code or even no-code mechanisms to design and implement application behavior – Flow Designer and UI Builder just to name a few.

A Flow is not just a single record, it is represented by a vast network of records, which are tightly coupled and are neither human readable as xml files nor can they be merged by a Source Control System in any meaningful way.

Another property of the ServiceNow platform is that an application is always connected to one source control repository branch at any given time. So, if two developers work on the same application on the same instance, they make changes to the same set of records that constitute that application.

One may argue about the design decisions made – but that won't change the fact that

- Many application files just CANNOT be merged in a Source Control System
- Multiple developers cannot work on multiple branches of an application on one development instance at the same time

To cut this short: **forget gitflow, forget feature branches, forget merging!**

With that being said, the questions are

- How can developers work on separate features?
- How can features be released selectively?
- What to do if the work on a feature is not yet finished when an application version should be shipped?

The answers to these questions can be found in chapter "Technical Debt Management" and – in more technical terms – in chapter "Handling Unfinished Work".

If feature branches are not an option – how should developers now work with branches? What value do we get from the Source Control System after all?

The Source Control System serves three main purposes:

- To keep a reversible trail of all changes deemed relevant by the development team
- To store all baselined application versions that were ever delivered by the development team
- To allow switching between different application versions – that may be multiple versions in development – and all versions that were ever baselined

This is how it works:

- Each application has its own repository (or "project" in GitLab's terms) – and this will stay that way even if ServiceNow ever decides to support storing multiple applications in one repository in the future
- The default branch in all application repositories is to be named "dev" (most Source Control Systems start with a branch called "master" or "main" when creating a new repository – these branches need to be deleted)
- The "dev" branch is always the active branch in the development environment
- Whenever an application version is baselined, a new branch is created based on the current state of the "dev" branch which is to be named exactly like the version to be baselined
- This new **version branch** is to be locked immediately after being created – no further changes to this branch should be allowed
- In case of a hotfix, the existing version branch is installed on an instance, the application is modified, and the new state is stored as a new version branch (which possible represents a lower version than the latest version already stored as a branch)
  The instance used to develop the hotfix depends on the instance setup used. Ideally there is a dedicated HOTFIX instance reserved for ad-hoc hotfixing. If such an instance is not available, hotfixes can also be created on DEV or TEST instances (see chapter "Instance Setups" for more details).

By applying these rules, the read-only version branches form the audit trail of delivered application versions by the development team and the single source of truth of anything that will ever be deployed on a downstream instance. Note that applications can only be installed from a branch (not from a tag) hence branches must be used.

Developers will always commit their changes to the "dev" branch during regular development.

The guidance should be that developers commit all changes made on the development instance as soon as possible – even if they are yet unfinished and not yet ready for deployment.

In theory, a team could also commit only finished work and stash any uncommitted changes during application baseline – when a new branch is being created – and then reapply the stashed changes after switching back to the dev branch. Real world experience with two larger developer teams revealed that – although technically feasible – the approach is more difficult to manage and more error prone when executed.

For hotfixes, the process is slightly different – please refer to chapter "Hotfixing and Backporting".

If a development team currently works on a version of an application that is soon to be released (e.g., version "1.x") and at the same time the next – and backward incompatible version should be worked on (e.g., version "2.x") – a second development instance is required. The work on this

future version would then be stored in a second "dev2" branch – please also refer to chapter "Hotfixing and Backporting" for further guidance on that scenario.

The list of branches of an application could look like this:

- dev
- dev2
- 1.0.0
- 1.1.0
- 1.2.0
- 1.2.1
- 1.3.0
- 2.0.0
- 2.1.0

To summarize: ServiceNow application files cannot be merged in a Source Control System and hence any method involving feature branches (e.g., gitflow) is not an option. To allow for selective feature deployments and the ability to always provide new application versions from a "dev" branch, features must eventually be turned off and treated as technical debt.

The branching strategy can hence be described best as "commit at head" as all changes are to be committed to the "dev" branch as soon as possible – no matter if the related work is completed.

Application Version Baseline Procedure

After a sprint, or whenever a new feature has been completed – or a new version of a dependency application should be deployed – it may be a good point in time to release a new version of an application.

So, whenever the development team considers the state in the application's "dev" branch ready for a deployment, the following activities should be conducted:

1. Run the Code Sanity check on the app. Ensure the app has no findings. Fix findings if necessary and start over.
2. Run all available ATF tests – and fix issues if any tests fail and start over.
3. Insert or update technical debts and adapt switches to turn features on or off to match the product owner's expectations and decisions regarding
    a. Unfinished work
    b. Features not (yet) to be released to users
    c. Deprecated features
    d. All other forms of technical debts
4. Insert or update dependencies of the app (i.e., update the minimum version requirements).
5. Document all changes relative to the previous version in the application's release notes.
6. Review and update feature descriptions in the application manual
7. Set the application version in the application settings (if not done already).
8. Click on the "Baseline" UI action on the application form (if the DevTools open-source application is installed). This will repair several application files which may be modified by the users or even by the platform by mistake.
9. Commit all uncommitted changes to the "dev" branch.
10. Create branch and name the branch exactly like the version of the application (e.g., "1.5.0").
11. Switch back to the "dev" branch.
12. Lock the new branch in the source control system
13. Increase minor version number (and optimistically assume that the next version will deliver a backwards compatible new feature)
14. Add the postfix "WORK IN PROGRESS" to the application name
15. Add a new section in the application release notes
16. Commit all changes to the "dev" branch

Deployment Automation

Each deployment of an application version to a downstream instance should be conducted exactly according to the following procedure in exactly that order:

Automated:

1. Establish the dependency list based on the requested application version

2. Check availability of the requested application version branch in the Source Control System and all application version branches of the dependency applications

3. Check which application versions need to be deployed to the target instance

4. Check if there are any version incompatibilities between the to-be-installed versions and the already installed versions on the target instance
(e.g., if a version 1.5.0 is to be installed but version 2.0.0 is already installed – this would imply a version incompatibility and the deployment cannot be done)

5. Deploy all applications (in the order of the "dependency list") to the target system

6. Verify compliance to the Coding Guideline on all deployed applications using Instance Scan

7. Run installation scripts (in the order of the "dependency list")

8. Run all automated tests of all deployed applications (in the order of the "dependency list")
(if there is a "Platform" application available, run all automated tests based on the dependency tree of the "Platform" application. Refer to chapter "Application Architecture and Dependency Management")

Manual:

1. Perform smoke test

2. Optional: Clone production instance to HOTFIX instance

To perform these steps, the following technical capabilities need to build – as these are not provided OOTB:

- Establish the dependency list based on version branches in the repositories
- Identify all tasks necessary to conduct the deployment – including installing applications from source control, running Instance Scan checks and running ATF test suites – and execute these tasks automatically
- Run all installation scripts remotely on the target instance
- Ensure that there is only one deployment taking place on one target instance at any given point in time (otherwise a mix if different or even conflicting version states may temporarily impair the results of automated tests being run in parallel)
- Warn users of potential data deletions if an application no longer contains tables or columns in other tables which are present on the target instance

The open-source applications "DevTools" and "Deployer" support all the required technical capabilities.

Technical Debt Management

A technical debt is any state in a software module that has a negative impact on its maintainability and requires future work but does not prevent production use.

What exactly should be considered as a technical debt is certainly subjective. Here are some examples that can be considered as technical debt:

- Dependencies to outdated or even deprecated libraries or components
- Duplicate code
- Dead or unreachable code
- Incorrect or misleading names of database entities (tables) or fields
- API naming inconsistencies (e.g., functions "GetInstanceURL()" vs. function "GetRecordUrl()" – note the different spelling of "URL" vs. "Url")
- Non-compliance to the coding guideline
- Insufficient or missing documentation
- Insufficient test coverage
- Deprecated features
- Unfinished work
- Fully operational features not yet to be exposed to users

A lot has been said and written about the long-term costs and effects of technical debt. For the context of this process, there are two main take-aways:

- Anything that is identified as a technical debt must be represented by a backlog item and treated like any other item that represents e.g., a business requirement – apply the boy scout rule – not all technical debt is known (or even considered as such) when it appears – the whole team should always document technical debt when they see it
- Technical debts should be documented as part of the application documentation which ideally is part of the application itself (e.g., as a UI page).

From the above list there are two special forms of technical debts: Unfinished work and features not yet to be released. One might argue that these have no negative impact on the long run, the opposite is true. Only after a feature is completed, or a completed feature is finally exposed to users, it will generate value to the business, and not just incur costs.

However, in the context of this process, it is pragmatic to treat such cases as technical debt, because they need to be managed in the same way. Unfinished work obviously requires future work and a completed feature that does not provide value to users still needs to be maintained until it is released – which again may imply additional costs – although hopefully not for a very long time.

The cases of unfinished work and completed features not yet to be released are specifically important. The guidance in this document is to commit all work into the development branch, as early and as often as possible. This leads to a state in which the development branch may contain unfinished code or application files at the time of the next version baseline.

Since we advise not to use selective commits or feature branches (see "Branching Strategy") we need a technique to manage unfinished work (see chapter "Managing Unfinished Work").

The following outlines the life cycle of a technical debt:

1. A technical debt is identified – either by the development team, a product owner, or any other stakeholder

2. The technical debt is documented in the application documentation

3. The technical debt is added to the backlog

4. If applicable a switch is introduced to render parts of the application inactive or at least invisible to users. The switch is set to whatever is appropriate at the time.

5. The affected application is baselined any number of times with the set switch state

6. Once appropriate the switch is flipped and the feature becomes visible (or invisible) depending on stakeholder's decisions

7. The affected application is again baselined any number of times

8. The technical debt is resolved, the switch is (and eventually the feature) is removed from the application

To perform Technical Debt Management on a day-to-day basis, the platform must be extended with the following capabilities:

- Document technical debt of an application in a way that these records are part of the application so that they are also stored in the application's source code repository
  - The technical debt records must be stored in a way that they can be retrieved and processed from a source code repository
- Documentation on technical debt can be generated automatically based on these records
- Each technical debt record has a switch that can be turned on and off
- A function is available to request the state of a technical debt switch – to control application behavior depending on the switch state

The latter two requirements are crucial for cases of unfinished work and features not yet to be released to production.

These requirements are not met by the ServiceNow platform OOTB – hence these additional capabilities must be implemented.

The open-source application "DevTools" supports these capabilities.

Hotfixing and Backporting

A hotfix is the resolution of a defect which is applied to an application version that is already deployed to a non-development environment – i.e., to a test or even production instance.

Backporting is the process of implementing the same or a comparable resolution to the current state of an application in a development environment.

When looking at the quality gates described it seems as if the delivery of new applications is a one-way street: development of a new feature or a bugfix starts in a development environment. These new features and bugfixes are then shipped to test and later to production environments as part of a baselined application version.

But what if a defect is identified in production? The defect may be severe and may be causing incidents. Users may be affected, and a business process may come to a halt.

A fast solution is required.

Two scenarios are possible:

1. The application has not changed significantly in the development instance since it was last baselined, tested and deployed to production.
   Any changes made in development since the latest released version can all be set inactive (refer to chapter "Handling Unfinished Work").
   The application can be tested with minimal effort – and most of the relevant test cases have been automated.
   Only few or no dependencies exist.

2. It's complicated

In scenario 1 a bugfix can be implemented in the development environment, a new application version can be baselined, deployed to the regular test environment, tested, and deployed to production – all of that within an hour or even less. However, this is not how the real world looks like most of the time.

Scenario 2 is the one we need to look at.

What if the application has changed significantly since the last version baseline?

What if multiple applications need to be changed to apply the bugfix?

What if other applications depend on these applications?

What if one or more affected applications are being worked on now and a lot of unfinished work has been produced and committed?

What if one of these applications is even backward incompatible to the version in production?

What if multiple versions of these applications have been baselined in the meantime and the state in development differs significantly from the version currently in test and the version in test differs from the version in production?

What if resolving the issue in production would take days or even weeks if the regular process is followed.

This is when the **Hotfixing and Backporting procedure** comes into play:
Please note: the "HOTFIX" instance is the instance that is used to produce the hot-fixed application versions (refer to chapter "Instance Setups" for details).

1. A defect is identified in production

2. Optional: a clone is made from production to the HOTFIX instance (ideally this is done after each production deployment)

3. Verify all application versions installed in the hotfix instance match the application versions installed on production (just in case a deployment took place after the most recent clone)

4. The defect is reproduced in the HOTFIX instance

5. The affected applications are identified

6. A decision is made whether the defect can be resolved according to the regular process – in that case the procedure stops here, and the regular process is applied. If in doubt, continue with the "Hotfixing and Backporting" procedure.

7. For each of the affected applications, a new branch is created named "hotfix"

8. The defect is fixed in one or more applications in the HOTFIX instance

9. Optional: ATF tests are created to verify the resolution of the defect

10. Testers and other stakeholders test the fix and perform a regression test on the HOTFIX instance

11. The resolution is confirmed

12. New application versions are baselined directly in the HOTFIX instance – however instead of the "dev" branch, the "hotfix" branch is used – refer to chapter "Application Version Baseline Procedure". This includes running all ATF tests and Instance Scan checks.

13. The new version branches are directly deployed to production
    (not to the TEST environment as it may be in a state that is too different from production to get reasonable results)

14. A smoke test is performed on production

15. Stakeholders confirm the resolution of the defect

16. The defect resolutions are backported to the current state of the affected applications in development
    (see chapter "Backporting")

17. ATF tests are also backported to the affected applications in development or new ATF tests are introduced to verify the resolution of the defect

18. Optional: New Instance Scan checks are introduced to prevent similar mistakes in the future if applicable

19. If the affected applications are being developed in multiple development environments – e.g., because an application must be worked on in the current and a future major version at the same time – the resolutions and ATF tests need to be backported to these future development instances as well

It is advisable to configure the Backlog Management system to support this procedure in addition to the regular process – as the steps are significantly different.

Further it is advisable to identify a few individuals in charge to manage this procedure, to conduct specific training and perform simulations. The procedure is time critical and will likely not be used very often – so the team's capability to perform this procedure must be kept alive proactively.

For technical considerations on backporting, please refer to chapter "Backporting".

In theory, the procedure may also be applied to defects identified during a regular test conducted on a test environment. However, if a team decides to apply the procedure in that case, this should be seen as a symptom of yet another severe problem: test cycles are obviously too long!

Release Notes

No one likes to write documentation. Or do you? Who reads that anyway?

Wherever the creation of documentation can be automated, based on the artefacts in the platform, that is great. However, there is one form of documentation that a human being will need to write – unless we have even better versions of generative AI – and these are release notes.

Release notes describe the changes made between the last baselined version and the current version.

Each application must have release notes. Within a development eco-system, release notes are ideally all at the same place or follow a common pattern so that they can easily be found by other developers, product owners, operations, and other stakeholders across teams.

But why are they so important?

Because others may depend on it. Literally. If an application A has a dependency to application B, application A must specify the minimum version requirement for application B. If new features are being added to application B on which application A depends on, the developers of application A must update their version requirements.

Some developers are lazier than others. The laziest developer might think: "I just update all the minimum version requirements to the current versions found in the development instance. This way I always depend on the latest versions, and I do not have to read release notes or communicate to anyone."

If everyone would behave that way, no application could ever be deployed after baselining. Any new baselined application version would then depend on various other application versions that are not yet baselined and hence it would not be deployed unless all other developers provide newer versions of their applications to the source control system.

When baselining an application version, the developers need to update the **minimum** version requirements to their dependencies, but they should do it wisely. To identify the minimum version requirement (and not just the latest) they need to know which capabilities already existed in versions baselined earlier – and the single source of truth for that are (or should be) the applications' release notes.

Developers are well-advised to write release notes immediately when they finish the work on a change to their applications – the text written for the release notes can also be used as part of the commit message – laziness may work in our favor after all.

Backlog Management

Every development and deployment process requires a backlog management system – a tool where a team can track the requirements and the progress of their work. Chances are high that your organization already has one – or many.

Some use Jira, others use GitLab, Excel, paper sheets on a wallboard or ServiceNow. No matter which tool you use, to support this process, some configuration is required.

In some organizations, this is a challenge. If many teams are using a single instance of a backlog management system, the department that operates the system tries to standardize the workflows across the organization and avoids customizations for individual teams or projects.

Seen from a high altitude that makes perfect sense. From the perspective of a team that tries to get work done in the most efficient way - this is just a headache.

If you are one of the fortunate and you have a say in how your backlog management system should be configured, you may find this guidance helpful.

The backlog management system should support the following states for a backlog item:

1. **Draft / Need more info**: The requirement description is being worked on or more information is required to implement the backlog item.

2. **Ready**: The author of the backlog item claims the item contains enough information to start development. A developer should be assigned to the backlog item to verify that sufficient information is contained in the backlog item.

3. **In Development**: A developer works or is asked to work on the implementation of the backlog item in the DEV environment.

4. **Done / Committed to repo**: The developer has completed their work and committed their changes to the source code repo - which makes the backlog item part of any subsequent deployment to the TEST environment.

5. **In Test**: The changes related to the backlog item has been deployed to the TEST environment and a tester is asked to test the result.

6. **Test Failed / Showstopper / Back to development**: A backlog item has failed the test and must be considered as a showstopper - so the current version on the TEST environment MUST NOT be deployed to the PROD environment. The backlog item is sent back to a developer who is now asked to fix issues. It is important to distinguish this state from "In Development" because the current deployment on TEST should not be deployed to the PROD environment if there are showstoppers. If a test failed but the issues are not considered as showstoppers, the version can be deployed to PROD, and the state should be set to "In Development" which indicates that they do not represent a reason to delay deployment to PROD but still need to be worked on.

7. **Test Passed / Ready for PROD**: The test has been passed and the backlog item is ready for deployment on the PROD environment.

8. **Review on PROD**: Deployment to PROD environment has been completed. A final check should be done before the backlog item is marked as "Completed".

9. **Completed**: The backlog item is finally completed.

10. **Cancelled**: The backlog item will not be implemented or worked on any further.

Ideally each backlog item, once it enters the state "In Test", should be linked to an application version – as some state transitions for one backlog item might imply the state transition for all other backlog items that are covered in an application version.

It is extremely helpful – although that is not mandatory – if the backlog management system also contains a list of personas that can be associated to backlog items. This allows to search and filter for backlog items that refer to specific personas and help testers to align their activities when testing multiple backlog items.

The backlog management system should also have a list of components which can be associated to a backlog item. A component can be a scoped application but also other things like the mid server, an external mail server, documentation assets, etc.

That makes it easier to group and filter for backlog items relevant for a component.

If your backlog management system is also able - based on default settings for each component - to assign developers and testers – either individually or as a team – depending on the state of the backlog item, that is even better.

All these capabilities can be built in systems like Jira or ServiceNow (e.g., using the "Agile Development" product) – but this requires changes to the OOTB configuration.

The open-source application "Agile" supports many (although not all) of these capabilities.

The "Agile" open-source app does not support sprints and epics – and this a conscious choice.

Whether the idea of "sprints" really add value to a development process is debatable. If you read this, you are likely open to radical ideas, like organizing your development team not in sprints but based on a prioritized backlog only.

Even more debatable is the idea of a category system that distinguishes between epics and stories. Many teams struggle with the question on whether a backlog item is an epic or a story, how large stories may be until they should be split into smaller stories – and whether this turns the original story into an epic and if they should have sub-epics or sub-stories or story-tasks. Many teams have come to the point where they gave up and realized that any multi-layer category system just follows the golden rule of category systems: No matter how it is designed, it turns out to be wrong at some point.

After all, it does not make any difference whether you call backlog items "backlog items" or "stories" or "issues" or "tasks". The author of this document believes that the popularity of the term "story" results from a misconception about agile methodologies mixed with the idea of "user stories" to describe requirements - which influenced the terminology used in Jira.

A backlog item is agnostic to that. Any backlog item in the "Agile" app can depend on other backlog items – and hence form a tree of related backlog items that can be put into a definitive order.

At the end of the day, we all do tasks one after the other. No matter how we call them.

The main takeaway is that the Backlog Management tool should be configured to reflect the states described to support the process in the best way possible.

Instance Setups

Any instance setup that at least consists of 3 instances can support the proposed process – however to allow parallel version development and independent hotfixes, a more complex setup is required.

But first things first: here is a taxonomy for instances:

DEV The main development environment where the next version of an application is being worked on.

DEV2 A secondary development environment where a future version of an application is being worked on. Working on DEV2 should be the exception. Changes made in DEV should be deployed as fast as possible – to reduce the amount of finished work waiting for deployment (which by the way may even have a negative CAPEX effect).
However if backward incompatible changes are required, these changes may require to work on them without interfering with current development.

TEST-AUTO The test environment on which ONLY automated tests are performed – this includes Instance Scan checks and ATF tests. An application version that does not pass all tests on this instance should not be deployed to any downstream instance. A TEST-AUTO instance mainly protects the TEST environment from too many – unnecessary – changes. Especially in environments with large test teams this allows for better planning and more stability in the TEST environment.

TEST The main test environment where testers and stakeholders perform manual tests and confirm that the installed release package is fit for purpose.

TEST2 A secondary test environment to confirm fit-for-purpose of new future versions – which are typically developed on a secondary development instance, and which may have backward incompatible changes to their current major versions.

PRE-PROD An instance that is as close to the production instance as possible to verify the deployment of a release package before deployment to production.

HOTFIX This environment is kept in sync with production and is on stand-by to allow the development and test of a hotfix version of an application version that is already deployed to production. Ideally the HOTFIX environment is cloned from production after each production deployment.

PROD The production instance where regular users perform their work and value is generated for the business.

These names can be referred to as the **"pipeline names"** of the instances (as opposed to their technical or domain names).

The following chapters propose three different instance setups with varying complexity (and costs).

Default

3 instances: DEV, TEST and PROD

This is the typical setup for many ServiceNow clients. It does not allow for parallel development of future versions and does not support hotfixing.

This renders the overall process less sophisticated but if the number of dependencies is low and complexity of application changes is low, this setup is sufficient.

All automated tests should be carried out on the TEST instance.

Balanced

5 instances: DEV, TEST-AUTO, TEST, HOTFIX, PROD

This setup adds the hotfixing and separate automated testing capability to the process. Since the setup allows for quick defect correction on PROD without interfering ongoing development and testing work – the setup is more robust and recommended if the frequency of deployments is low, but the complexity of the changes is higher – which makes it more difficult to baseline new application versions from the current state on the DEV environment.

Juggernaut

8 instances: DEV, DEV2, TEST-AUTO, TEST, TEST2, HOTFIX, PRE-PROD, PROD

This setup supports all aspects of the process: parallel development on multiple major versions of applications (for backward incompatible changes), parallel testing operations, hotfixing and an additional safety layer through an additional deployment test on PRE-PROD.

This setup is recommended for high complex application eco-systems with numerous dependencies between these applications and a mix of applications that are changed at a low or high frequency.

Making a Choice

The following overview should help to decide on which setup fits best:

	Default	Balanced	Juggernaut
Required Instances	3	5	8
Deployment Frequency	Multiple deployments per **hour**	Multiple deployment per **day**	Multiple deployments per **week**
Separate automated testing	NO	**YES**	**YES**
Hotfixing and Backporting	NO	**YES**	**YES**
Simultaneous development on future application versions	NO	NO	**YES**
Additional deployment verification	NO	NO	**YES**

Techniques

Scoped and Global Applications – and App Customizations

The most critical technique (to be learnt) is to ship all features, all customizations, all changes, as an app – and the scoped app is the preferred choice.

Before we can investigate how that works, a myth must be debunked: Yes, scopes have been introduced as a security boundary – to limit access to sensitive data to a specific set of users. But scopes have evolved into something different. With the introduction of source control support, scopes are in fact a packaging boundary. A source control system's repository would contain all application files associated to a specific application – hence the role of the scope as a security mechanism has been superseded by the role of a packaging mechanism. For the means of the process described in this document, a scope cannot be both. Its role as a security mechanism for custom applications must be ignored, its role as a package mechanism must be pronounced.

The chapter "Cross-Scope-Scripting" contains the details on how to "ignore" the security aspect of scopes.

The ask to ship everything as an app is less of a statement what to do – it tells us more about what to avoid as part of a deployment:

- Capture and deploy changes in Update Sets
- Changes to OOTB records
- Changes to records in plugin or store application scopes
- Changes in the global scope
- Manual XML exports and imports of any sorts
- In fact, ANY manual activity as part of a deployment

Instead, all necessary changes should be shipped as part of an application. "Application" can mean:

- Scoped Applications
- Global Applications

For the sake of completeness, also "App Customizations" should be mentioned. "App Customizations" allow to make customizations – i.e. add and modify records in a store app or plugin scope and capture these customizations in a source control repository.

The App Customization is always bound to a specific version of the underlying customized application – that prevents it to be installed over an incompatible version.

However, using App Customizations comes with several downsides:

- App Customizations are not applications – i.e. they are NOT derived from sys_app – that means that any technical capability around applications that is required to support this process would have to be created in two flavors: for genuine apps and for app customizations (e.g., dependency management, installation scripts, deployment automation etc.).
- App Customizations are "highlanders" – there can only be exactly one per scope. If different capabilities are to be built "on top" of a plugin, all of these must be

implemented in the same App Customization – this reduces flexibility and limits options on how to design the overall application architecture.

- When upgrading the underlying plugin or store app any customizations must be migrated (pretty much like any change to OOTB records) which can be a laborious and time-consuming process.

With all these downsides using App Customizations should be limited to cases where any other form of shipment for whatever reason is downright impossible. It should be the ultima ratio. A scoped application with installation scripts acting across scopes any OOTB record can be deactivated and replaced by a modified copy shipped as part of the scoped application.

	Scoped Application	Global Application	App Customization
Scope and namespace	Application specific	global	Same as customized plugin
Supported in source control	YES	YES	YES
Namespace	All application files contained in the application share the same namespace	All contained application files share the same namespace as all other records in the global scope and those contained in other global applications	All contained application file share the same namespace as the records in the plugin that is customized
When to use	Always as a default	ONLY as a fallback	ONLY if complex customizations already exist

As a general rule, scoped applications are ALWAYS the best choice. Global applications or customization applications should ONLY be used if complex changes to records in global or plugin scopes have already been made at the time of the introduction of this technique. In this scenario the efforts and risks of refactoring the same features into scoped applications may just be too high.

The following cases should be addressed through installation scripts:

- New or modified ACLs are required where scope boundaries prevent their use
- New records where a requirement exists that these records must be in a specific OOTB scope (global or plugin)
- If many records are affected and such changes are very complex, consider the technique described in chapter "Data as Code"

In summary: scoped applications are always the best and preferred choice to transport changes of all kinds, including changes to records in other scopes.

One application making changes to other applications may appear risky and uncontrollable. As these changes are part of an application's defined and repeatable behavior - which can be validated and tested - the risk is acceptable.

This implies a fundamental change in how many changes are done in the ServiceNow platform. The implications (both technical and organizational) of this guidance should not be underestimated.

Application Architecture Design

Implementing the idea of shipping everything as an app leads to more questions in practice. And these are different questions than most ServiceNow development had to answer in the past.

The most important question regarding the application architecture is: what capability, what "file" should be in which application?

How should the superset of "everything" be distributed into applications?

The challenge is there is no simple answer for that and certainly no one-size-fits-all approach. As so often in software development – or any other complex challenge: it depends.

Architecture in this context means to solve an optimization problem with conflicting goals:

1. Applications should be small – so that they are easy to understand and to maintain

2. The number of applications should be limited – so that it is easier to keep track of them

3. Applications should have as few dependencies as possible – so that they can be changed and deployed independently

4. Code duplication should be reduced – so that a defect does not have to be fixed in multiple places

The conflicts are obvious

- Smaller applications lead to more applications, which leads to more dependencies, which leads to more complex deployments
- Larger applications lead to artefacts that are more difficult to maintain
- Applications maintained in different teams lead to more code duplication (yes, it does)

There are some guardrails that can help making these decisions:

1. One team (within a larger development department) should be responsible for one or more applications

2. Applications should only serve a single purpose or "concern." Examples are:

   a. One process (that applies to both new custom processes as well as when extending OOTB processes like "Incident Management")

   b. A cross-process or even cross-platform capability like "base data management" or "data segregation"

   c. A specific user interface like a workspace for the ServiceDesk, a Portal for end users, a mobile user interface for a specific purpose

   d. Once particular system to connect to (A "Spoke" in ServiceNow terms)

   e. Re-usable components for a specific purpose

And there are special applications that further help structuring the application architecture:

1. The "Platform" app – the app that no other app depends on, but which depends on all other apps – this should be the one and only app that describes a deployment package

2. The "Core" app – the app that all other apps may depend on because it contains re-usable components specific to the instance.

The key element to this technique is that decisions on how to distribute capabilities into applications – and hence when to create new applications and how to name them – are made consciously and not ad-hoc. Any product owner may describe a new requirement and may have the power to set the priority for it. But this should not automatically lead to ad-hoc decisions regarding the application architecture. Every develop should be aware of the need and value of making conscious and considerate decisions and discuss the options with peers (or escalate the case to the architects) before just starting the implementation.

The Communities of Practice are an excellent forum to discuss and make such decisions. Discussing and deciding on such matters publicly fosters awareness and knowledge alike so that the likelihood of better decisions by every member of the team is increased over time.

Cross Scope Scripting

When developing scoped applications developers face several challenges. Some Glide API features are not available in scoped applications and some tables prohibit access from other scopes.

When trying to access assets across scopes chances are high that a developer sees one of the following error messages:

- "Security restricted: Access to api [some function] from scope [some scope] has been refused due to the api's cross-scope access policy"
- "Security restricted: [some operation] operation against [some table] from scope [some scope] has been refused due to the table's cross-scope access policy"

The good news is there are ways to overcome these restrictions.

Cross scope restrictions come in two flavors:

- The inability to read, create, modify, or delete specific records
- The inability to use some Glide API features because they are not made available to scoped apps

Whenever you create your own scoped app and try to access records which reside in any other scope than your own you may bump against the cross-scope restriction wall. Likewise, when you try to use several Glide API methods your script may produce errors indicating that the method is not available for scoped applications.

Prominent examples of Glide API features that are only accessible in global scope are

- GlideRecord::setWorkflow() – which is used to avoid business rules to be executed when updating or inserting a record
- GlideImpersonate – which is used to impersonate a different user – which is crucial in scripted ATF tests

To understand these restrictions, it is helpful to take a closer look at what scopes are in the first place.

Scopes represent a boundary around an application and its data. Anything within a scope has free access. But access to both data and code in a different scope is limited. So, are scopes mainly an access control mechanism?

Not really. At the same time, a scope is a packaging mechanism. That may not have been so clear when mankind still used update sets to ship their product increments from one instance to another. However, when following the guidance provided in this document, applications are the only means to transport code – and scoped applications are defined by their scope.

The scope is the natural boundary around a piece of versioned software that we ship.

Over time scopes have emerged into two things at the same time: an access control mechanism (for application code, not for users) and a packaging mechanism.

Both are valid intentions. We want to control access from one area to another. We also want to package software for deployment and re-use.

However, these two intentions lead into a conflict - as scopes try to do both at the same time. The package we want to ship and the security boundary we consider fit-for-purpose may not overlap.

This conflict becomes more obvious when we think of a scoped application - let's say an application that deals with highly confidential data. No other application shall be able to access data stored in any of the tables of that application. We appreciate the cross-scope restrictions that can be configured for these tables. All fine so far.

Now think of several customer specific adaptions and extensions to this application. We want to develop these additions in a separate package, give it a separate version and name. And we want to ship it separately. We can think of these additions as "plugins" or "extensions" to the original application.

It appears straight forward to implement these plugins as a new separate scoped application. But cross scope restrictions will make it impossible to access the table in the first application from the second. The packaging boundary and the security boundary are different - and in that scenario that is a challenge.

The best way to connect two applications would be for the first application to expose an API or extension point to allow other applications to extend the first application in a defined and structured way - however in many cases such APIs or extension points do not exist.

Since Quebec and Rome developers can create customizations to a store application which allows to create a new application that shares the same scope of an application that has been installed from the store. Since Rome this also includes installed plugins. However, this option is not available for applications that were created by customers or the open-source community.

As mentioned earlier yet another type of scope restriction applies to the Glide API. Some features are just not accessible if executed from a script that belongs to a scope other than global. These restrictions are - as many other features of complex systems - the result of many conscious decisions made over time:

At some point scoped applications were seen as something coming from the outside - built by clients, partners, community developers - not by core ServiceNow developers - and hence were met with suspicion. Applications - according to this line of thinking - are inherently insecure and as such should not have access to all the potentially dangerous privileged features of the Glide API - like impersonating users (through GlideImpersonate) or doing changes to records without triggering their business rules and other automated workflows (using GlideRecord::setWorkflow).

However, with the introduction of the "global scoped application" one can build applications that have their own scope as a packaging boundary, but which execute in the global scope and hence do not experience any API related restrictions. So, considering applications as dangerous per se is not the real issue. To create global scoped applications, you must add a system property to an OOTB instance first. Although openly documented, the global scoped applications remained to be unknown to many developers.

Another argument is that the implementation of the APIs for global and for scoped applications differs and that every API implementation must be adapted - mainly due the namespace implications - and that this work just has not been finished yet. Some of the restrictions were removed over time - so more and more Glide API features become available in scope - as the API evolves over time.

Conversely there are API features which are not communicated to be available to scoped apps although they are. One example is GlideUser::getMyGroups() which is documented in global but undocumented in scope.

Some of these API limitations are seen as an enforcement of good practices when developing applications. Whenever a scoped app is using features from the global Glide API that should be

done with great care and be the result of a conscious decision. Developers should consider both security, architectural, deployment, performance, and governance aspects.

At the end of the day, scopes remain to be two things at the same time: a means to package a set of assets as a versioned software module and a security boundary.

To make this process work – the security boundary needs to be ignored – as developers need full cross scope access for their applications.

How to cross scopes?

Cross scope restrictions apply when code running in scope tries to access data from a different scope and when code makes global-only API calls.

Tables can be configured to restrict access from other scope per operation. One can define whether create, read, update, or delete operations should be allowed from other scopes or not. That is the granularity one can define. Not more, not less. Which means that in the example of one application "extending" another application, the only choice is to open the table with the highly confidential data to all other scopes and not just to the application that serves as a trusted plugin of the first application.

In addition, modifying the configuration of these tables - which might be part of OOTB features, plugins, or 3rd party apps - represent a change to assets that may be updated at some point in the future - and that creates more complexity when upgrading. That should be avoided.

As mentioned, in an ideal world the first application provides a defined API that other applications can use to implement extensions - but in many cases that API is not available.

We should keep in mind that these restrictions apply on a script level. They control what one script can do with data in a different scope. What users can or can't do is fully controlled by ACLs. That is a totally different story.

So, configuring a table to allow cross scope access may not be what you really want if access should only be granted to specific other applications.

As stated before, access to tables from one scope to another is only a part of the issue. Not being able to execute all Glide API functions from a scoped application is the other. So, what we need is a way to execute code in either global scope or a scope of choice from within a scoped application.

If a script could be executed in a scope of choice that script would also be able to avoid the table-based scope restriction, since the create and modification functions of GlideRecord will work fine if executed in the same scope as the protected table.

How to execute any code in any scope?

An obvious option to execute code in a different scope than the application that needs it, would be to create a script include in the corresponding scope. But that's not really satisfying since the aim is to ship application as one package - and the scope of your application serves as exactly that packaging boundary. To ship applications AND additional script includes in different scopes in additional global scoped apps or (even worse) as additional update sets should be avoided.

If additional script includes that need to be shipped are not an option, what else can be done?

A Glide API class comes to the rescue: GlideScopedEvaluator. This class allows to execute a script that is stored in a field of a record in any table - if the record is in the same scope as the

caller. This record however does not have to be in the database. It is sufficient if this record is in memory, represented by a new GlideRecord object.

There is one precondition for the GlideScopedEvaluator class to work across scopes:

The global system property "glide.record.legacy_cross_scope_access_policy_in_script" must be set to "true".

Before doing that, we need to keep in mind that this will enable any script from any application to execute code in the global and any other scope. This may or may not be considered as a security risk. So, check back with the responsible governance body before making that change.

The following code example of a single function Script Include that illustrates how a scoped application can use the GlideImpersonate class which is only available in the global scope:

```
function ImpersonateUser(strUserSysId)

{

        var grSI = new GlideRecord('sys_script_include');

        grSI.get([THE SYS ID OF THIS SCRIPT INCLUDE BUT COULD BE ANY]);

        grSI.script =

        "var glideImp = new GlideImpersonate(); result = glideImp.impersonate(user+")+";";

        grSI.sys_scope = strScope;

        var evaluator = new GlideScopedEvaluator();

        evaluator.putVariable('result',null);

        evaluator.putVariable('user', strUserSysId);

        evaluator.evaluateScript(grSI, 'script');

        return evaluator.getVariable('result');

}
```

The code example demonstrates how it works – however a clear recommendation should be made to implement a single re-useable function that makes use of the GlideScopedEvaluator class – so that all scripted cross-scope activities are channeled through this single function. This allows to control and audit such cross-scope activities.

Not using the GlideScopedEvaluator anywhere else may also be part of the Coding Guideline enforced by Instance Scan.

As a summary: When developing scoped apps developers may run into cross scope restrictions as the required security and packaging boundaries of a scope may not overlap. This is particularly likely in an application's installation script – as changes may have to be made in different scopes than the application scope. Using the GlideScopedEvaluator class in combination with the system property "glide.record.legacy_cross_scope_access_policy_in_script" set to true is a solution to that.

The open-source application "DevTools" contains the technical capabilities to execute code from any scope in any scope. Refer to the functions RunScriptInScope() and RunScriptInGlobalScope().

Installation Scripts

Installation scripts aim at making modifications to a target instance that for whatever reason cannot be transported as application files through source control and hence cannot not be deployed in regular ways.

This includes the following cases:

- System properties which need to be set to different values on different target systems (e.g., logging verbosity levels)
    - Changes to OOTB global or scoped records (e.g., deactivating specific OOTB UI Actions or Business Rules, or modifications to ACLs)
    (The benefit over shipping these OOTB records as part of a global or customization app is when such records are updated as part of a family or store release – in such cases all updated records can be accepted and the installation scripts can be re-executed after the upgrade)
    - Creating records in other scopes than the application scope (e.g., ACLs on tables defined in other scopes)
    - Modification to records that are in tables not derived from sys_metadata – and hence are considered data
    - Creation or modification of users and groups and assignment of roles to such groups and users (e.g., technical users required to run Scheduled Jobs etc.)
    - Configuration of Scheduled Jobs – as they need to be associated with a user to run as
    - Configuration of (existing) database views
    - Creation of database indices
    - Modifications to other applications – however that must be limited to downstream dependencies

To perform such changes, an installation script needs to be built in a way that it

- is shipped as part of an application
- is only executed ONCE as part of the deployment of an application – after the application has been copied to a target instance and static code analysis checks have been performed, but before any automated functional tests are run
- is idempotent – which means that even if it is executed multiple times on the same instance, the result is always the same (so it does not create duplicate records etc.)
- can make changes to records that are either in no scope (i.e., represent data), in the scope of the application or even in any other scope
- ideally performs all modifications through a specialized API that allows the execution of the script to not actually do any changes but to create documentation on what is intended to do (however that is optional)
- complies with a specific naming convention and/or interface definition so that it can be executed as part of an automated deployment process
- returns success or failure so that an automated deployment process can respond to errors

This set of requirements implies some unexpected super-powers and constraints:

- An installation script MUST NOT be shipped as a fix script – since fix scripts do not allow any control on whether or when the script is executed when regular OOTB application deployment mechanisms are used. Fix scripts do execute immediately after copying an application to a target instance – e.g., through source control "Apply Remote Changes" – and this is not acceptable
- An installation script must be able to make changes to records in other scopes and eventually access Glide API functions that are only available in global scope – hence cross scope scripting techniques are required (refer to chapter "Cross Scope Scripting")

As with any other feature or capability installation scripts must be tested. The following ideas should be considered:

- Specific ATF tests that verify the state of records after running an installation script
- ATF tests that verify the application's behavior where the underlying logic depends on the successful execution of the installation script
- The execution of an installation script within an ATF test and the subsequent verification of changes made during its execution

These ideas can be combined or used in different scenarios – there is no one-size-fits-all solution.

The open-source application "DevTools" contains all technical capabilities to

- Run scripts in any scope from any scope
- Perform all changes through a specialized API
- Execute installation scripts for an application and all its direct and indirect dependencies

The open-source application "Deployer" allows to execute installation scripts on a remote instance for integrating with "Deployment Automation".

Instance Specific Configuration

As the title suggests, some configuration needs to differ between different instances – that is different records, settings, values, options on a DEV, TEST, PREPROD, PROD or other instance.

Such configurations may represent:

- Credentials and connection configuration pointing to connected systems – where a DEV environment may not be connected at all, the TEST instance is connected to a mock environment of a remote system and the PROD instance is connected to the PROD environment of that remote system
- Configuration on outgoing emails – e.g. to prevent emails to be sent to real users from within DEV and TEST environments
- Configuration on user sessions – like its duration, idle times, etc.
- Log levels
- Security configuration
- Titles, logos and colors in the GUI to indicate to users which instance they are currently using
- System properties of all kinds

The use cases of instance specific configuration may be obvious, the answer to the question how to deal with them in the deployment process is not.

Should such instance specific configuration be deployed as part of one or more applications?

Or should such configuration be applied and managed manually on each target instance explicitly.

As often in complex environments, the answer is: "it depends".

The following options exist:

- Manual changes on each individual instance
- Changes made on DEV (or any other instance considered as a reference) and then manually exported and imported to downstream instances
- Deployed as part of applications – overriding any deviating value on the target instance
- Deployed as part of applications – maintaining any existing value on the target instance – thus allowing later manual changes

With these options at our disposal, for each individual use case a different choice can be made – but no matter which choice is made – it should be consistent.

Development and operations teams must decide and agree who "owns" specific types of configurations and who is responsible to maintain it and how.

Configuration that is deployed as applications should be "owned" by the development team as they direct changes made to the target instances through the applications they ship.

Configuration that is maintained manually on each target system should be "owned" by the operations team.

When instance specific configuration is deployed as part of an application the installation script is the most obvious place to check for the instance to which the configuration is applied and control the specific values and settings depending on the instance.

It is recommended to add such agreements into the coding guideline and implement corresponding Instance Scan checks to enforce it. This can at least prevent developers from accidentally setting configuration values that are in fact "owned" by operations.

However, using technical names of instances (i.e. the DNS or URL names) should be avoided as the applications otherwise become dependent on the specific technical infrastructure on which they run.

Take the following scenario of a company called "ACME" which has 4 ServiceNow instances (in the ServiceNow cloud as it seems):

Role	URL	Technical Name	Pipeline Name
Development	**https://acmedev.service-now.com**	**acmedev**	**dev**
Test	**https://acmetest.service-now.com**	**acmetest1**	**test**
Pre-Production Test	**https://acmetest2.service-now.com**	**acmetest2**	**test**
Production	**https://now.acme.com**	**acmeprod**	**prod**

Note that the URL and the technical name (as found in the pseudo system property "instance_name") sometimes differs – maybe for historical reasons?

Imagine, an instance (no matter whether DEV, TEST or PROD) is re-installed, physically, or logically moved or - for whatever reason - renamed – e.g. because the company is acquired and obtains a new name. In such a scenario, applications would have to be refactored to use the new technical instance names in the corresponding installation scripts.

Instead of using URLs or technical names to identify an instance during installation, applications should use so-called "pipeline names" instead of "technical names" in installation scripts.

A mapping is required between technical names ("instance_name") and the pipeline names – and this mapping must be made available to all applications to be used in installation scripts.

To support this technique the following capability needs to be implemented:

- A mapping between technical names and pipeline names – which can be maintained independent from applications to be deployed
- Installation scripts can query the pipeline name of the current instance

The open-source application "DevTools" contains all these technical capabilities ("GetInstancePipelineName").

Data vs. Code

What is data and what is code? And why is it important to differentiate?

To get this question answered, we need to get the terminology in order. This document describes a process by which applications are developed in a development environment and then deployed to other instances. For the sake of this question, we will consider everything that is shipped as part of one of these applications as **Code**.

Data on the other hand is whatever is created or modified by users, administrators, or integrations in production or what is created or modified in sub-production instances by testers, developers, test integrations for the sake of testing.

One might argue that this differentiation is arbitrary and irrelevant in the ServiceNow context. Code (in this wider sense) which may include anything from "real" code artifacts like a Script Include to all sorts of "Configuration" (like the selection of fields visible on a form) is stored in database tables. From a pure technical point of view there is little difference between an incident, a user, a group, a company, or a configuration item ("Data") and Script Includes, Business Rules, a Decision table, or a UI Page – all of them a stored as records in one of the many tables.

There are however important differences – because we treat Code and Data records differently in this process – and we need to define different people (or roles) to be responsible and accountable for them.

Here is a collection of examples – by far not conclusive – to ensure the reader can spot what is Code and what is Data going forward:

Code	Data
The Application record itself	Users
Script Includes	Groups
Business Rules	Users assigned to Groups
UI Builder Pages	Roles assigned to Groups
Client Scripts	Companies
Catalog Items	Tasks / Cases / Incidents / Problems / Changes
Portal Pages and Widgets	Attachments
Tables	Emails
Fields	Configuration Items
Roles	Events
Credential Aliases	Credentials
Connection Aliases	Connections
Scripted Web Services	

As a rule of thumb, the following distinction works well:

Code is anything stored in a table derived from sys_metadata or sys_scope.

Data is anything stored in a table NOT extended from sys_metadata or sys_scope.

Code is transported as part of an application through the process described – it is always changed in a development environment
(Note that Code that is modified by Installation Scripts on downstream instances remains Code).

Data is maintained directly on the various instances through some kind of operational process in production or as needed for testing purposes on sub-production instances.

That concludes that **Code MUST NOT depend on Data**.

(E.g., this means that a Flow must not reference a specific group for a task assignment directly. The group should rather be configured in an instance specific system property and be read by the Flow.)

So far so good. There are however several grey zones where the distinction is not that obvious:

Scheduled Jobs

A scheduled job is stored in the sysauto_script table which is extended from sys_metadata.

OOTB the table is excluded from being captured in source control through the attribute update_synch=false. Let's explore why:

The Scheduled Job record contains information when and what to run. The "what" is a script – clearly, that is Code.

The "when" is debatable. Maybe a Scheduled Job should run in different intervals on a sub-production instance than on the production instance. Maybe on the development environment it should not run automatically at all – but only when the developer wants it to run – for testing purposes.

The exact orchestration of many Scheduled Jobs running during the day, or the week should be a decision to be made by administrators responsible for the production instance and should not lay in the hands of developers who might not know enough about the full set of jobs that need to run. So, developers may not be able to make informed decision for when their job should run.

Even worse, Scheduled Jobs also contain a reference to a user that should be used as the execution context for the Job. As users are clearly Data, how can this be? Code must not depend on Data, so how can a Scheduled Job (Code) contain a reference to a User (Data) and hence be dependent on it?

Despite some research no conclusive answer could be found by the author. It just is what it is, and it cannot be fixed easily due to the need for backward compatibility.

But the question remains: how to deal with Scheduled Jobs? They are neither clearly Code nor are they clearly Data.

There are various options:

- Consider Scheduled Jobs as Data and NOT include them into applications to be deployed. Administrators must then add Scheduled Jobs manually on each instance – and eventually copy the to-be-executed code from documentation provided by developers
- Consider Scheduled Jobs as Code and
    - change the attribute on sysauto_script to update_synch=true and
    - add installation scripts that
        - add technical users for the execution on target instances automatically
        - configure the Scheduled Jobs to use these dynamically created technical users
- Consider Scheduled Jobs as Code and ship them without associated technical users and delegate the responsibility to configure technical users and timing to administrators. Such configuration could eventually with the help of a separate application which is focused on instance specific configuration, but which is maintained by administrators.

In this case no clear guidance can be provided as the best suited solution depends on the organization, the existing separation of duties and applicable security policies.

The option where the applications create their own technical users is appealing as it is pragmatic and requires little coordination between teams – but some organizations may just not accept users being created (and security privileges being assigned to them) by applications as part of their installation scripts – as such an activity may strictly fall into an operations team's responsibility.

Other scenarios can be identified in which the distinction between Data and Code is not that clear, where OOTB structures do not support this clear differentiation or where Code may rely heavily on specific data structures.

An example for such a scenario is product definitions – which technically are Data, but which may only change as part of a corresponding Code change and hence may have to be treated like Code. Refer to chapter "Data as Code" for guidance on how to do that.

It is important however to make conscious decisions on how to treat these cases explicitly and define responsibilities and technical processes to create, update and delete such records.

Data as Code

As pointed out in chapter "Data vs. Code" there are scenarios in which certain data records should be treated as code.

A prominent example are product definitions as used by the ServiceNow product Order Management for Telecommunications and other industry workflow products.

Product definitions contain – as the name suggests – information on complex offerings with many variables and variations. Imagine a large telecommunication company offers network services to large clients. A product definition in that context may consist of hundreds if not thousands of individual records.

A design decision has been made by ServiceNow to consider these product definitions as data – so that users can modify these product definitions in production to adapt them in day-to-day operations. This implies that all code must be designed to work on any potential constellation of data.

An equally valid approach is to consider such complex product definitions as code that is shipped as part of applications. Imagine there are strong and complex dependencies between product definitions and integrations with 3rd party systems, user interfaces and the logic to process orders based on these product definitions.

This is just one of many examples where the boundary between data and code is blurry and there are good arguments for either considering records as data or to consider them as code.

The question is: what if there are records in tables that are NOT derived from sys_metadata - and hence are in fact data records – but these records should be shipped as part of an application – and hence should be treated as code?

How to ship data as code?

Any application can define new tables that are derived ("extended") from sys_metadata. This means that all application can contain any kind of data which is stored in records in such extended tables. In this document such tables are referred to as "transport" tables.

To include data records in an application, a mechanism must be created to copy such data records INTO a transport table. This mechanism can be used by developers at any time when appropriate – the latest point in time would be during baselining the application. During deployment the records in the transport tables are installed to the target system (as part of the application) and would then be copied into the actual data tables. This can be done in installation scripts.

There are mechanisms available on the platform OOTB that allow to include individual records as application files. However, these OOTB capabilities are not fit-for-purpose to ship large complex networks of records.

To support this technique the following capability needs to be implemented:

- A scoped application defines a transport table that stores data records' metadata and content
- A mechanism that copies data records stored in the transport table into the original data tables on a downstream instance (to be used in installation scripts)
- Both mechanisms should support idempotency to avoid duplicate records

The open-source application "DevTools" contains all these technical capabilities ("Record Transporter").

Managing Translations

Some aspects of the ServiceNow platform are a bit confusing – one of these aspects is how translated text is stored and maintained.

There are multiple tables that explicitly contain translated text

- sys_ui_message
- sys_translated
- sys_translated_text

There are numerous tables that contain text visible to users which do not have any specific attributes to specify the language of the text. Examples are:

- sysevent_email_action
- sysevent_email_template

In total there is no consistent mechanism to store – and hence to deploy translated text across the platform.

To gather insights on the coverage of translations for a given language compared to a base language (e.g., English) is a form of art. Guidance on how to manage, maintain, test and validate translations could be subject of a separate whitepaper.

The guidance in this document is focused on the deployment aspect.

Most tables that contain translated text are derived directly or indirectly from sys_metadata – so these tables contain Application Files – which are regularly captured into a Source Control repository and are deployed in the same way as any other Application File. For those records, no special treatment is needed.

However, there is an outlier: The table sys_translated_text.

Any table can contain string (or HTML) fields that are "translated". The content of such fields is English by default but if a user selects a different language for their session and modifies the contents of such a field, a record in sys_translated_text is created for that specific language.

A record in sys_translated_text refers to the record, the table, the field, and the language for which the field was edited.

And here comes the challenge: As stated, any table can contain a translated field – that includes Application Files (i.e., tables derived from sys_metadata) – so a user (a developer) may add translated text to a record that is an Application File and which is part of an application to be deployed.

The mechanism of translated fields is universal, so sys_translated_text records can both refer records that are **code** (Application Files) and to records that are **data**.

The table sys_translated_text, however, is NOT derived from sys_metadata – i.e., these records are NOT Application Files – and hence not captured as part of an application and not committed to the Source Control System.

If the intention is to ship translated text for fields contained in an Application File (e.g., the name and description of a Catalog Item stored in table sc_cat_item) – the corresponding records in sys_translated_text – which are technically **data** records - must be treated as **code**.

In order to support this, the same principles can be applied as described in chapter "Data as Code".

In addition, a capability must be implemented to identify all sys_translated_text records that are associated to Application Files contained in a specific application.

The open-source application "DevTools" contains all these technical capabilities (Refer to function "RecordTransporterPackagePackTranslatedText").

Test Driven Development

"By writing clean code that works and demonstrating your intentions with automated tests, you give your teammates a reason to trust you." Kent Beck, *Extreme Programming Explained*.

This chapter provides an overview on good practices when writing ATF tests and maintaining an ATF test suite that provides value on the long run. Moreover, the author of this document clearly advocates the practice of test-driven development – or short "TDD" – which implies that tests should be authored as soon as possible – which may even be before the first line of production code is written.

As we all love lists, meet the 10 rules to follow to build and maintain a test suite that provides value over time:

1. **Ship tests as part of apps**
   Each application should have a default test suite which is named exactly like the application. This test suite is then be used to validate the application during development, before baselining and during deployment to non-production instances. All tests that validate an application's behavior should be part of the suite and be part of the corresponding application.
   This way code and tests always stay in sync.
   The application's tests are always part of that application and will also be deployed to production – where there will probably never run – unless specifically designed.

2. **Create all required test data within the test**
   Tests must not have any dependencies to data, the environment, specific configuration. All tests must be able to always pass. That implies that any data that is required to simulate a behavior must be generated as part of the test itself.
   This may seem cumbersome in the beginning, especially when complex data structures are required to simulate a use case – e.g., think of an incident related to a CI which is impacted by other CIs which is follow-ed up by a problem that later results in a change. Many teams struggle with the creation and maintenance of test data on development and test instances. By creating code that creates test data, not only automated tests can make use of it – but also manual testing.
   The platform's Automated Testing Framework undoes any data changes performed during an ATF test, which includes the creation of test data as part of a test. So, creating test data as part of the test also ensures that test data does not clutter the instance over time.

3. **If a behavior can be tested without the GUI, don't use the GUI**
   It is tempting to use existing manual test plans and procedures as a template to create automated versions of it – and thus just trying to replace human testers performing scripted manual tests by a robot. However, this approach does not produce any value. Resulting tests are complex, depend heavily on the GUI and are costly to maintain. Complex manual tests often try to validity many different aspects of an application's behavior into one single test – for practical and efficiency reasons. When creating automated tests, the tests should focus on the narrowest scope possible – and often a specific part of the functionality of an application can (and should) be tested without using the GUI.
   This sometimes requires refactoring code.
   E.g., imagine an UI Action that performs a complex operation on one or more records. To test this behavior without the GUI, the code may have to be moved into a Script Include first, and the UI Action script only calls the function in the Script Include, and eventually

provides feedback to the user.
Such a function could then be tested without the GUI by creating the necessary test data, then calling the function directly from a test script and then validating whether the data was manipulated in the expected way.

4. **Verify one behavior in each test**
Each test should focus on one behavior or functionality only. A failing test provides useful information to developers. If only one specific behavior is validated in a test, it is easier to track the root cause if tests have a narrow scope.

5. **Prefer many smaller tests over few larger tests**
To narrow the scope of tests inevitably leads to many small tests. And that is a good thing.
Big is bad, small is good. There is no balance to be kept on this one.
Is it more difficult to maintain many smaller tests? No, it is not. Any change, intended or unintended, may cause tests to fail. If a change is intended, and tests must be updated to reflect the new, modified or introduced behavior, it is again easier to modify many smaller tests, then modifying larger more complex tests.
Smaller tests also help better in understanding the real impact of a change.

6. **Use mock interfaces instead of real external systems**
Tests should have no other dependency but the code and logic they are validating. This also includes external systems. If data is imported or ingested from an external system, the investment in mock interfaces or mock data objects almost always pays off. It may be difficult and cumbersome in the beginning, but as soon as the foundation is laid, any addition test scenario can be added quickly – thus increasing test coverage at a high pace.
External systems may not be connected to all instances and not all external systems may support interaction with automated tests.

7. **Avoid using asynchronous processes**
Whenever asynchronous processes are involved tests may become flaky – that is for no obvious reasons they may fail sometimes. This is frustrating and annoying. One of the most frequent reasons for flakiness are tests that trigger asynchronous processes – and that must wait for the process to finish to evaluate the result. Sometimes this hits a timeout, and the test fails.
Refactoring code in a way that allows to execute the functionality directly as part of the test – without the asynchronous processes around it often helps to create both more robust code and more reliable tests.

8. **Make conscious choices on (GUI-based) end-2-end tests**
While stating that the GUI should be avoided, does not mean that it should not be used at all. Each application may have a very small number of automated GUI based end-2-end tests. However, such tests should be the exception and the test cases should be chosen with great care. The intention of such tests should also explicitly validate the GUI itself – and not the underlying functionalities.

9. **Prefer scripted tests over separate test steps**
The Automated Testing Framework supports various GUI-based and server-only test steps that can be used to assemble complex test procedures. Each test step can then end the test with a failure if a defined condition is not met.
In theory this works well – especially for GUI-based tests – but for unit tests – that is the validation of individual JavaScript functions or classes this approach is not helpful.

Whenever it is feasible, use scripted tests that contain only a single "Run Server Side Script" test step.

Within this script, create the necessary test data, execute the code to be tested and validate its result.

When different test cases – that is different combinations of test data and function parameters – should be tested, it is strongly advised to add these test cases into a single script.

The tests will execute faster, as test data constellations may be re-used and do not have to be created (and removed) for every single test case.

Another advantage of multiple test cases in a single script is that if one or more of these test cases fail, the test script can continue to validate all other remaining test cases within the script. If the test cases were implemented in separate scripts, the first failing test step would stop the test – and less information is available for a developer to investigate issues.

10. **Re-use test code using Script Includes**
The Automated Testing Framework supports so-called Test Step Configurations – these are re-usable scripts that can be integrated into an ATF test. Experience with larger teams working on complex applications shows that maintaining re-usable test code in Script Includes is easier to handle and such re-usable test code can be executed more selectively from within other scripted tests.

Wiring the inputs and outputs of an ATF Test step into other scripts requires additional effort and does not provide any real value – and it requires to split the test code into several script steps.

The most common re-usable test code is a function that generates test data. If implemented in a Script Include the same function can also be called outside the ATF test – e.g., to prepare test data to be used in a manual test.

The open-source application "DevTools" contains capabilities to
- Write clean unit tests using the TestAPI class
- Create test data using the TestDataAPI class
- Create new tests in a test suite
- Run tests from within test step and test result forms

Semantic Versioning

Versions are not just numbers with dots in between. Versions convey an important message. They indicate progress.

There are different ways how version numbers can be used. To support the process described in this document best, version numbers should be "semantic".

Semantic Versioning means that version numbers indicate whether the difference between two versions of a software module (an application in ServiceNow's terms) are

- Only bugfixes
  AND/OR
- New features
  AND/OR
- Contain backwards incompatible changes

A semantic version tells developers and users what to expect from a new version. They manage expectations. And they indicate required activities – especially if an application depends on a technical interface exposed by another application. If a new version indicates a backward incompatible change, it may mean that developers of the application that depends on the modified application must adapt to these changes so that their application continues to function properly.

A semantic version is a prediction of compatibility expressed by the team that sets the version.

A prediction however is not a guarantee. The responsible developer team might make a mistake – but once the version is set and the application is baselined – the only way to fix it is the release a new version that expresses the actual compatibility correctly.

Semantic versions are hence crucial to set expectations and to communicate the need for further changes in other applications in a larger developer community.

This is how a semantic version is formed:

Given a version number MAJOR.MINOR.PATCH (e.g., "1.0.0")

Increment the…

MAJOR version when incompatible API changes are made (e.g., "2.0.0")

MINOR version when functionality is added in a backwards compatible way (e.g., "1.1.0")

PATCH version when backwards compatible bug fixes are made (e.g., "1.0.1")

The use of semantic versioning has been widely adopted in the open-source community and many commercial software projects and products.

But that is not always the case.

Many companies use versions to indicate a release plan or even use version changes for marketing purposes. The change of the major version in such cases rather indicates a significant number of new features, a new design or just the fact that the marketing department wants to draw attention to a product.

It might just feel like an understatement that hundreds of new features have been added in version 1.1.0 following version 1.0.0.

At the same time, it might appear overdone to increase the major version if there is only a single, small change done to an API. Imagine an application exposes a REST API with hundreds of endpoints and in one single endpoint, a single parameter name is changed – a tiny little change - almost unnoticeable. But even that single small change breaks the contract between the API provider and its consumers. And the developers of the applications that depend on that API must check their code and eventually adapt their code to use the new parameter name.

Semantic versions communicate compatibility. They do NOT communicate the number of changes.

A detailed description of semantic versioning can be found here:

https://semver.org

Backporting

The chapter "Hotfixing and Backporting" described the scenarios and use cases in which backporting comes into play – and how this activity can be handled from a process perspective.

The following chapter provides insights on how that can be done on a technical level.

Given there are two development states of the same application for whatever reason:

1. One state where a change (often a bugfix) has already been applied
   (the "source state")
2. A second state where the same change needs to be applied as well
   (the "target state")

The "same change" in this context does not mean that the exact same modification of code or records needs to be made.

The "same change" means that the same specific outcome should be achieved – but sometimes the same outcome can or has to be achieved using different modifications.

Backporting may be necessary in case of

- Hotfixing
- Working in multiple development streams – where multiple (in most cases major) versions of a single application are in development simultaneously

For the sake of the description of technical options in this chapter, we use the hotfixing scenario as an "case study" – but the techniques described can be applied to all possible backporting scenarios.

Backporting is always a manual process. Forget any approaches that mechanically identify changes or differences on a record level and then copy them from one instance to another. That may work in some corner-cases, but most of the time, it will not.

There are several techniques that may play a role in backporting:

- Copy selected sections of code from a record in the source state to the same record in the target state
- Copy selected sections of code from the source state, insert AND modify that code in the target state
- Implement new code in the target state
- Remove existing code in the target state and re-implement the feature in the target state
- Deprecate the affected feature in the target state and declare the next version baselined from the target state as a backwards incompatible change

"Code" may refer to actual JavaScript code and any form of low-code, no-code or other configuration assets contained in an application.

Which of these techniques should be applied depends on

- The changes that have already been made between the source state and the target state
- Whether the already implemented change to the source state is sufficient to achieve the same outcome on the target state
- How robust the change to the source state is

There are different ways to transport the code from the instance with the source state to the instance with the target state:

Selected sections of JavaScript Code or other pure text-based assets:
Copy-paste through a workstation's clipboard

- For limited configuration changes on a limited set of records:
  Re-do the changes manually in the GUI on the instance of the target state
- Where one or more records from the source state can be used as-is in the target state:
  Update Sets or XML exports

(Yes, that is the only scenario in which Update Sets may play a role.)

Handling Unfinished Work

In the chapter "Technical Debt Management" it is established that "unfinished work" is one out of several cases that we refer to as a technical debt.

This chapter describes in more detail how the development team can handle unfinished work in the code.

In an ideal world the technical debt record is being created in immediately when the work on a new feature starts – if it is to be expected that the feature will not be completely implemented at the time of the next version baseline.

In the real world, most teams won't be able to predict with certainty that this situation may occur. There may be a plan for the next version baseline – but changes in priorities, upstream dependencies, or an urgent bugfix might require a version baseline earlier than planned. As a result, it is difficult to predict if the work on a new feature needs to be documented as a technical debt and if a switch needs to be built into the new feature.

Does it make sense to create a technical debt in all cases and even build in switches for every new feature? Certainly not – each team will need to develop a sense on whether it is necessary and if so when.

The latest possible point in time is when the application version baseline should take place. If the plan is to have a sprint review on Friday morning and to perform an application version baseline in the afternoon – a team might spend the lunchbreak with building switches around a new feature that they expected to deliver, but the product owner just did not consider to be ready. Obviously, that is not what we want.

There are certain aspects that a development team should consider when designing new features in the first place:

- Build new features as if there was the requirement to be able to switch them on or off
- Split the code and configuration to be created into smaller pieces

In addition to the code design, a team can also reduce the likelihood of having to manage unfinished work by carefully planning when which feature is to be built and by whom.

Please refer to chapter "Reducing Unfinished Work" for more details on how to avoid having to manage unfinished work in the first place.

One of the technical requirements for day-to-day technical debt management described in chapter "Technical Debt Management" is to provide a function that can be used to query the state of a technical debt switch.

This function can then be used to control if a feature should be active and visible to users.

Here are some specific examples on how to implement such switches:

- Script Includes – use the function directly in code
- UI Actions – use the function in the condition script field
- Client Scripts and UI Scripts – Either set the active flag to false/true or add logic in the client sided script – however this requires a display business rule to transport the switch state from the server to the client.
- UI Policy – Set the active flag to true or false
- Table - Configure advanced read ACLs and use the function in the scripted response – note that ALL read ACLs for the table must implement the logic
- Fields – Use the active flag to true or false
- Flow – Either publish/unpublish the Flow or create an if statement querying the technical debt record directly

For all types of application file records where it is not feasible to add scripted logic and use the function, but which have an active flag – or any other field that controls whether the application file is active – one can also set these fields as part of the installation script of the application (see chapter "Installation Scripts").

Reducing Unfinished Work

Unfinished work are application files in an application that are still work in progress.

Unfinished work should be avoided for three reasons:

1. When baselining an application, unfinished work needs to be managed – i.e. switches need to be introduced and later removed, documentation must be maintained – see chapter "Handling Unfinished Work".

2. When there is urgent demand for a defect correction, any unfinished work makes the fast baseline of an application more difficult – thus increasing the time (and efforts) to defect resolution. This is not only because any unfinished work must be handled on short notice, but the state of the application may have changed since the version for which the defect correction is needed has been baselined.

3. Any work that has been done should yield value to users as fast as possible. Anything that has been done but is not used, increases capital costs – and that is bad for the business.

Unfinished work can however be reduced through the following tactics:

- Split code into smaller chunks – and prefer single function Script Includes over of larger classes (where the various functions do not share any states and the class only acts as a namespace). It is easier to finish and extend smaller functions.

- Move complex code logic to Script Includes instead of keeping code in Business Rules, UI Actions, Flows, etc. This way, unfinished code is less likely to impact application behavior. This does not necessarily reduce unfinished work per se, but it reduces the likelihood that it must be managed as such.

- Plan the work on multiple backlog items impacting a single application within the development team so that there is as little overlap as possible and aim at finishing one backlog item off before starting the next.
  On the other hand this may slow down a team if team members would have to wait for each other.

- Baseline more often. Even if not all work has completed (or even started) within a sprint creating an application version baseline – even if there is no plan to deploy it – reduces the likelihood that unfinished work interferes with already completed capabilities. This may, of course, consume some time which could otherwise be used for development work.

- Refactor the application architecture to split capabilities into more but smaller applications to reduce the likelihood of unfinished work at the time of the baseline of a single application. That in return increases the number of applications to manage and the dependencies between them. So, a balance needs to be found.

Treat Evaluator Warnings as Errors

The Evaluator is the component in the ServiceNow platform that reads, interprets, and executes JavaScript code. Whenever the Evaluator fails to continue executing code a warning is logged in the syslog table, with the source field set to "Evaluator".

The execution model of the ServiceNow platform is robust. Errors encountered by the Evaluator do not crash the platform or in any way hinder the platform from continuing to function. But a script that runs into an error will just stop on the spot. This may lead to an undefined state if data was modified in the part of the script that already ran, but this data modification was not completed as the script stopped.

The fact that a failing script does not produce any visible, obvious error message and is "just" logged as a "warning" in the syslog table conceals the fact that every single Evaluator error must be treated as a critical bug that must be fixed. Not more, not less.

To ship applications that are fit-for-purpose and work as intended it should be self-evident that such errors should be tracked and fixed before an application is baselined and presented for further deployment.

As stated above these errors remain invisible unless a developer explicitly searches for them in the log. This leads to many of such errors to remain unnoticed – even though hundreds, on some instances thousands of such errors are being logged every day.

The fact that even OOTB components produce such errors on a regular basis does not help either. Some developers even interpret the fact in a way that even ServiceNow does not care about these errors too much (in fact they are logged as "warnings") – and conclude that such errors are not relevant at all.

The opposite is true. Any script that fails due to syntax errors, the use of undefined variables, or other reasons that lead to a sudden and unhandled stop of a script can cause significant damage. That damage may pile up silently in small drops over days, weeks, and months. The outcome can be functional issues, but it may also be compliance or security problems.

A mature development and deployment process should consider to

- Treat the Evaluator "warnings" as errors – because that is what they are
- Surface such errors to developers to make them aware of failing code in the first place
- Establish a zero-tolerance policy against such errors – e.g., if such errors are being logged during a test, the affected applications must be corrected, and the test must be considered as failed

The open-source application "DevTools" contains all technical capabilities to

- Display recently logged Evaluator errors to developers (while hiding Evaluator errors that were caused by OOTB components)
- Convert Evaluator warnings to errors

Personas and Roles

Managing access to specific users or groups of users can be confusing. It is helpful to agree on terminology in the first place:

A **role** (in the ServiceNow context) is a type of configuration that links users or groups to specific access rights (like the write access on record of a specific table, a UI Action, or access to a specific REST API endpoint). A role is a technical asset – a record stored in the sys_user_role table.

A **persona** is a description of a group of human beings (or an individual). This description is provided from a business perspective. A persona is agnostic to any technical implementation. Some personas may not even interact directly with a system (specifically with a ServiceNow instance) and yet be very relevant to a business process.

The tricky part is to transform requirements which usually contain references to personas into technical configuration and logic.

As a "Help Desk Agent" (the persona) I want to be able to see all open incidents.

The OOTB role "sn_incident_read" provides read access to incidents, so assigning the role to whatever group constitutes the "Help Desk Agents" sounds like a good solution.

It is generally accepted good practice NOT to assign roles to users directly but to groups instead.

Let's assume there is one (or more) groups that contain "Help Desk Agents" so assigning the sn_incident_read role to these groups appears reasonable.

This approach raises several questions:

1. Is the fulfillment of the above requirement targeted at the development team or at the operations team? Assigning roles to groups is a data change, not a code change.

2. How can the fulfillment of this requirement be tested?

3. What if "Help Desk Agents" are not only supposed to read incidents, but also to modify them – and do many more things: shall all the different roles that are required then be added to these groups?

If the answer to the first question is: the requirement is targeted to the development team and hence the implementation should materialize as some form of code (in a wider sense) then the approach to assign the specific role to the group cannot be the solution.

After all, code must not depend on data. But what is the alternative?

Introducing the concept of the **persona role**. A persona role is a role that represents a persona. Such persona roles are named after the persona and contain whatever specific other roles required to allow (and deny) access to the users that belong to that persona.

As every other role it should not be assigned to individual users but to groups.

In the example the "Help Desk Agents" would be represented by a "persona_helpdesk_agent" persona role (which is part of an application) and the persona_helpdesk_agent role would then contain sn_incident_read role (and probably also sn_incident_write and many other roles).

Persona roles should be prefixed by the application scope name followed by ".persona_" – e.g., "x_yourprefix_core.persona_helpdesk_agent".

The implementation of personas (as an expression of business requirements) is now represented as a code asset which is fully testable with no dependency to production data.

If there is no documentation or concept of personas available yet, here is a list of usual suspects to start with:

- Business User
- Business User Team Manager
- ServiceNow Administrators
- Service Desk Agents
- Service Desk Supervisors
- IT Service managers
- CIO
- CIO Staff
- IT Expert (Fulfiller)
- IT Expert Team Manager
- HR Business Partner
- HR Business Partner Team Manager

Technical Users

Technical users are widely used. Scheduled jobs run in a specific user context – and organizations are well-advised NOT to link scheduled jobs to real user accounts which may be deactivated or deleted once the person leaves the company.

It is at least noteworthy that the platform does not have a specific representation of technical users in its OOTB data model. No "type" field on "sys_user" nor an extended table from "sys_user" or anything alike.

There are however the two fields "Web service access only" (web_service_access_only) and "Internal Integration User" (internal_integration_user) which are often associated with technical users.

This makes it at least difficult to even identify technical users if there is no policy, consensus, or convention within a team on how to setup technical users.

The following convention worked well for several teams for several reasons:

- A technical user has either field "Web service access only" or "Internal Integration User" or both fields set to "true"
- A technical user's "First name" is ALWAYS "Technical"
- A technical user's "User ID" ALWAYS starts with "technical."
- A technical user's "Last Name" contains a hint on what the technical user is intended for (e.g., "Integration SAP", "Performance Analytics", or "Incident Cleanup")
- A technical user's "User ID" ALWAYS ends with the lowercase, dot-separated equivalent of its "Last Name" (e.g., "technical.integration.sap", "technical.performanceanalytics", "technical.incident.cleanup")
  Note how the dot is used to express a hierarchy in the naming scheme.

By following that convention, platform operators can

- Filter for technical users
- Identify a technical user as the source of a log entry or any kind of errors
- Prevent that technical users are used in interactive sessions
- Apply automated processes to technical users
- Report on technical users (e.g., for security audits)

These considerations are important for the process for two reasons:

- As described in chapter "Data vs. Code" a conscious decision must be made whether technical users are to be treated as code. If that is the case, such users should be subject to a convention described in the Coding Guideline and ideally enforced through Instance Scan checks
- The Definition of Ready should define that a requirement description (in a backlog item) should define which technical user(s) are affected (created, changed)

Application Manuals and Release Notes

As always there are many feasible solutions, but the following pattern proved to be very effective for several teams:

Each application (scoped or global) should contain an application manual – and "contain" really means "contain". Technically the application manual is implemented as an UI page, which is part of the application, and which follows a naming convention (that considers both scoped applications and global applications).

Application manual naming conventions

- for a scoped application: "<scope_name>_manual.do"
- for a global application: "<unique_shorthand>_manual.do"

Since global applications not only share the global scope but also the global namespace for UI pages, a unique shorthand must be defined for each global application.

The contents of a UI page can be authored in plain HTML – or use functions that generate parts of the page.

Most application files (Script Includes, Business Rules, UI Actions, etc.) contain fields that can be used for documentation. The technical section of the application manual can be (partly) auto-generated by surfacing the data already contained in application files.

One section of the application manual should be the release notes.

Each version should be a sub-chapter. The sub-chapter title should contain the version number and the date of the version baseline.

It is recommended to define minimum requirements to the application manual in the Coding Guideline and enforce these through Instance Scan checks.

The open-source application "DevTools" contains all technical capabilities to

- render parts of application manual UI pages

The open-source application "CodeSanity" contains Instance Scan checks to

- Enforce that an application manual page exists

Organizing Documentation

This guidance refers to documentation usually maintained in a knowledge management system – like Confluence or any other wiki-style system.

No matter what system is in use to document the application, work procedures, goals, technical architecture, and all other aspects related to the operation of a ServiceNow platform – the challenges related to documentation are mostly the same:

- Specific information is difficult to find
- Information is often outdated and badly maintained
- Multiple, often conflicting versions of documentation exist

The following guidance helped improving documentation quality and documentation effectiveness for teams:

- Structure information based on the process roles: Information should not be structured by "topics" or by "the team that created it" but by the audience it addresses. Each process role (e.g., Developers, Product Owners, Deployers) has their own section.
- Each of these process role specific sections must have an owner.
- Each owner must review the available information in their sections on a regular basis – and rigorously remove anything that is out-of-date or no longer relevant to the team.
- Each process role specific section should contain a "need to know" area – where new team members can find all information they are required to know.
- Each process role specific section should contain a "how to" area – where practical guidance can be found on how to do things.
- The documentation around the communities of practice should also be visible on the highest level - since the participants of the communities of practice in most cases have various process roles.
- The Definition of Ready and the Definition of Done should also be placed on the highest level to be as visible as possible.

An example table content

1. Definition of Ready
2. Definition of Done
3. For Product Owners
    a. Onboarding / Need to know
    b. How to
        i. Author great backlog items
        ii. Prioritize backlog items to maximize business value?
4. For Developers
    a. Onboarding / Need to know
    b. How to
        i. Getting to done?
        ii. Scan application files?
        iii. Handle unfinished work?
        iv. Create unit tests for script includes?
        v. Create new CodeSanity (Instance Scan) Checks?
        vi. How to update open-source applications from GitHub?
5. For Deployers
    a. Onboarding / Need to know
    b. How to
        i. Prepare a deployment?
        ii. Perform a deployment?
        iii. Handle a failed deployment?
6. Communities of Practice
    a. Agile Backlog Management
    b. Architecture
    c. Development
    d. Deployment and Operations
    e. Manual Testing

Making the Change

### Pills and Surgery vs. Life Change

The process proposed in this document requires a substantial change. The change is technical. But that is the minor part. The bigger part of the change is about the change of people's behaviors.

As you can guess changing people's behaviors is the also bigger challenge. No new technology, no newly defined process or organization structure will yield any tangible results of people stick with their long-engrained, well-rehearsed routines.

When facing personal, or health problems, many people ask for pills and surgery instead of doing the hard work of personal development and changing their habits to change their life.

The same dynamic can be observed in organizations.

So, keep in mind: technology is secondary, it's all about people!
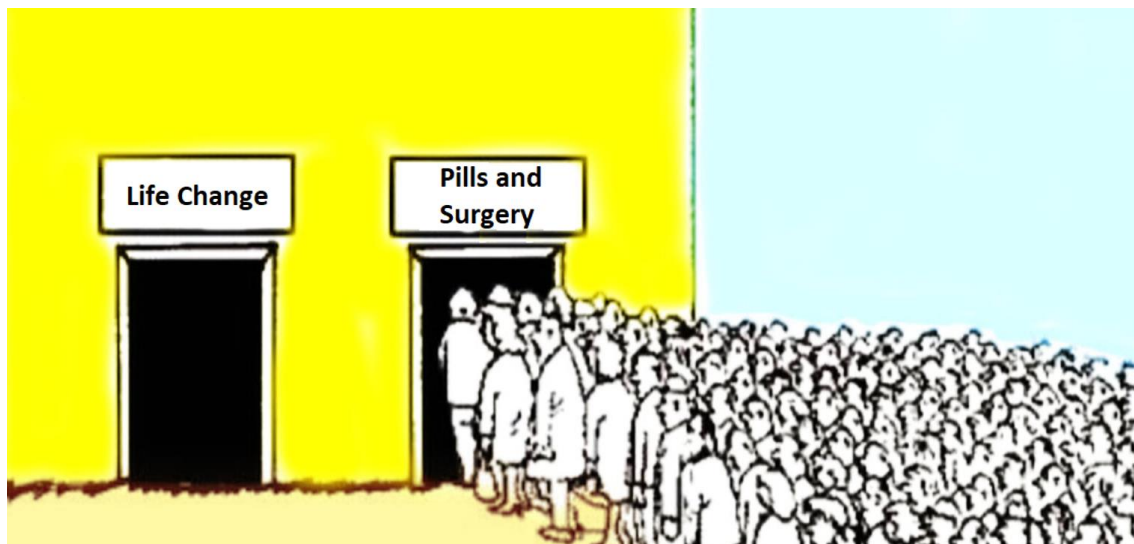
Getting out of the comfort zone

Now with that being said, it becomes clearer what the actual challenge is. It is not to write a few documents, get some technical capabilities in place, define the coding guideline, and set up some new rules for the team.

The actual challenge is to lead a team to a different way of working than what they used to. Things must be unlearned. Established routines must end. New techniques must be acquired, and new routines be rehearsed.

This forces people to get out of their comfort zone.

Different people will respond in different ways to these asks. Some might welcome the initiative with open arms, be fully supportive, creative, turn into champions and lead the pack.

Others may respond with criticism, cynicism, silent blockades, or even open revolt.

Be prepared to get to know co-workers, contractors, managers, and other stakeholders in a way you did not see them before – in a positive and negative sense.

Zero-Tolerance Mind-Set

At various occasions the guidance in this document is not to accept a single deviation from set rules. All code must be fully compliant with the coding guideline. Not a single automated test may fail. A deployment instruction for a release package must not contain a single reference to an update set or manual installation step.

This sounds hard – but it is crucial. Any deviation, any exception may have detrimental effects. If on one occasion it is ok to accept failing tests, then maybe it is also ok on other occasions, and then, what is the point to ask the team to write these tests in the first place?

It must be clear to everyone that established rules apply to all teams, to all individuals, in all cases.

These rules, however, be it the coding guideline, the definition of ready or any other guidance, must be malleable and the team should have a say in their definition. Ideally it's the team itself that writes, adjusts, and agrees on the rules. Only then can we expect full support for what is being asked.

Developer Onboarding

Being a member of the team while the transition is happening may be challenging, interesting, even inspiring. A developer that experiences the introduction of this process and its techniques from the beginning will have the time to learn and adapt to the new normal and most importantly they do so together with their peers.

A new developer joining the team after the transition has been made, however finds themselves in a different situation.

They do not only need to adapt to their new role, the team, the stakeholders, the context, the client's business domain but also must unlearn many things they know about how to do their thing in ServiceNow and instead lean new techniques.

This can – and in many cases will be – troubling and hence new team members need special attention. A defined and thoughtful onboarding protocol may come to the rescue.

The following list contains some ideas on where to start – many are independent of the recommendations in this document – but these are included anyway:

- Read this document – or a tailored version edited with the specifics of the organization
- Verify all needed access is granted
- Verify the developer is added to all relevant mail distribution lists and other communication channels
- Verify the developer is added to all relevant meetings
- Become a ServiceNow Certified System Administrator
- Become a ServiceNow Certified Application Developer
- Review the Definition of Ready
- Review the Definition of Done
- Review the Application Architecture Overview
- Review the Lighthouse Story (see the Definition of Ready, Definition of Done and the Architecture Principles implemented as a comprehensible example)
- Review the Instance Setup
- Review the Application Baseline procedure
- Review the Coding Guideline
- Create their own "lab" application on the dev environment, implement a small feature and make their application pass all automated QA checks (including ATF and Instance Scan checks)
- Pass a knowledge check

This list is certainly not conclusive, and each team and organization may have its own take and perspective on it. The most important aspect is that a new team member joining may face a bigger challenge to get up-to-speed than before the transition – as many technical and procedural aspects are very different from what they have experienced in earlier ServiceNow-related projects they have worked in.

Communities of Practice

Communities of practice are groups of people discussion options and making decisions on various topics. Consider establishing communities of practice (also sometimes referred to as "Chapters") for the following topics:

- Development / Code Guideline / Automated Testing
- Architecture
- Backlog Management / Agile Practices
- Manual Testing / Test Management
- Deployment and Operations

In the early days of a ServiceNow implementation, especially when the team is yet small, it is perfectly reasonable to combine these topics into a single community of practice – as the team grows it makes perfect sense to establish separate communities of practice for the different topics.

Each community of practice should meet at least once, better twice a week. The community of practice is a decision-making body, so the meeting cadence must allow to respond to urgent matters and questions and must be able to provide guidance and decisions on short notice.

Introducing the process proposed in this document is demanding. People need to learn new skills and adopt new ways of working. Communities of Practice are designed to make sure everyone has a say in how exactly this transition should be designed. There must be a place where concerns, objections, sometimes even fears can be openly expressed.

These Communities of Practice meetings should not be confused or mixed with the typical rituals of agile methods. These meetings should focus on how things are done, not to discuss new requirements or to review completed work.

The topics being discussed should be as relevant to the complete audience as possible. If topics are discussed in great detail, especially in a virtual meeting environment, many people may mentally walk away and perceive such meetings as boring and irrelevant. Moderating Community of Practice meetings is hence about maintaining a balance between going into details but also providing answers and decisions to specific questions.

The results and decisions made by the Communities of Practice should be respected and defended by management. Empowerment is key to the successful implementation of the process.

Moderating Community of Practice Meetings

Each community of practice is moderated by a facilitator who sets off the meeting, leads the sessions, and documents the decisions and outcomes.

The following techniques worked well for several teams:

- Documentation of the community of practice on one single wiki page containing
  a. "Waiting For"
     The "waiting for" list contains all topics that are work in progress and assigned to someone. "Waiting for" means that the community of practice is waiting for someone to come back with a result to make decisions
  b. "Backlog"
     The backlog is a list of topics to be discussed by the community of practice. Everyone can add new topics to the end of the list
  c. "Results and Decisions"
     Results and decisions are documented - with the latest entry on the top of the list. This helps everyone who may have missed a session to catch up.
- Invite everyone who may be interested in the topic
- Participation is strongly recommended but not mandatory
- Apply democratic processes in making decisions
- Let everyone be heard – especially the less "aggressive speakers"

The facilitator is well-advised to follow the following agenda in all sessions

1. Opening the session, welcome participants.

2. Ask for urgent topics that should be put to the top of the backlog.

3. Ask for status of "Waiting For" topics. If any if the topics is ready for discussion, it should be put after any identified urgent topics in the backlog.

4. Discuss topics from the backlog from top to bottom. Document any results in the "Results and Decisions" section or move topics to "Waiting For" if applicable. All topics moved to "Waiting For" must be assigned to a person or a group – otherwise the community will wait forever.

How to involve the less "aggressive speakers"?

It is difficult to get everyone's opinion heard – especially in a virtual meeting environment. In most teams there is a loud minority and a less vocal majority. This inherently causes a bias in the decision making. There are numerous techniques to involve participants that are less vocal:

- (Virtual) polls
- (Virtual) boards with sticky notes
- Breakout rooms

Backlog

The following list can serve as the foundation for a backlog for the project to introduce the process described in this document:

- **Educate** the whole team about the process as a whole and its building blocks
- Assign the different **gatekeeper** roles to individuals
- Educate all **gatekeepers** on their specific duties and responsibilities
- Define the **Backlog Management** process
- Configure the **Backlog management System**
  (e.g., by installing and configuring the "Agile" App or configuring an existing Jira instance available in the organization or by using the ServiceNow Agile plugin)
- Assign gatekeepers for each quality gate of the **Backlog Management** process
- Setup the **Community of Practice for Backlog Management** (which decides about changes to the Backlog Management process)
- Educate the team about the **Backlog Management** process
- Setup and structure the team's **Knowledge Management System** (e.g., Confluence)
- Define the **instance setup**
- Setup instances according to the defined **instance setup**
- Create and publish documentation on **abbreviations**
- Create and publish documentation on **terminology / glossary**
- Educate the whole team about **Application Architecture and Dependency Management**
- Implement **Dependency Management** (e.g., by installing the "DevTools" app)
- Setup the **Community of Practice for Architecture** (which decides on changes and extensions of the application architecture)
- Define, agree on, and publish the **Definition of Ready**
- Define, agree on, and publish the **Definition of Done**
- Define and publish the **Application Architecture**
- Inform the team about the **Application Architecture**
- Create the **"Platform" app** to act as the root node of the dependency tree
- Setup the **Community of Practice for Development** (which decides on changes to the Coding Guideline) – it should be a recurring meeting (1-2 times a week) where all developers and architects are invited
- Define a process for the **Community of Practice for Development** to make decisions if there is no consensus in the team
  (e.g., by democratic vote by majority or by delegation to the Community of Practice for Architecture)
- Define, agree on, and publish the first version of the **Coding Guideline**
- Implement automated checks to verify applications against the **Coding Guideline** (e.g., by installing the CodeSanity app)
- Educate the development team on **Automated Testing**
- Educate the development team about **Installation Scripts**
- Setup and configure a **Source Control System**
- Define the **Branching Strategy**
- Educate the development team about the **Branching Strategy**

- Educate the development team about **Semantic Versioning**
- Define and publish the **Application Version Baseline Procedure**
- Educate the team about the **Application Version Baseline Procedure**
- Define and publish the **Deployment** procedure
- Educate the responsible team about the **Deployment** procedure
- Implement **Deployment Automation** (e.g., by installing the "Deployer" app)
- Define and publish the process for **Technical Debt Management**
- Educate the development team about **Technical Debt Management**
- Define and publish the process for **Hotfixing and Backporting**
- Educate the development team about **Hotfixing and Backporting**
- Educate the development team about their obligation to write **Release Notes**
- Define and publish the **onboarding curriculum** for new team members
- Define and implement the **Lighthouse Story**
- Present the **Lighthouse Story** to the whole team
- Create a **Runbook**
  (e.g., by installing the "Runbook" app)

Self-Organized Learning

One recommendation to allow a team getting familiar with the process is to apply the technique of self-organized learning.

A learning facilitator is responsible to organize the process – they may also provide short introductions to the various topics – but ideally the topics are introduced by subject matter experts.

There are many different techniques to facilitate this. The following procedure worked well in both virtual and on-site environments:

1. Identify the topics to be covered (e.g., based on the chapters in this document)
2. Instructions from the Learning Facilitator
3. Create learning groups, identify group speakers, and agree on presentation dates – each individual team member should present at least once
4. Learning groups pick topics – the facilitator may have to decide which groups get which topic if some topics are not picked or multiple groups want to work on the same topic
5. Optional: Short presentation on the topic by an expert or guest speaker
6. Learning group members read / consume materials (i.e., chapters of this document)
7. Learning groups discuss and reflect the content within their group
8. Learning groups summarize the content (e.g., as a mind map or in other creative ways)
9. Summaries are shared with the facilitator who may provide feedback
10. Learning groups prepare presentations
11. One learning group member performs a presentation on their topic to the complete team
12. Discussion in the whole team
13. Feedback and further input from subject matter experts and the facilitator

Platform Capabilities

Automated Testing Framework

The Automated Testing Framework is an OOTB platform capability that allows creating and running automated tests. Tests consists of test steps which simulate user interaction or any other code execution on the platform. Each test step can conclude with a success or failure result. If the latter, the test is considered failed. Any data changes conducted during a test are rolled back after the test is completed.

Tests are organized in Test Suites. One Test can be part of multiple Test Suites.

All tests, their steps and Test Suites are application files and hence can be part of a scoped or global application.

Data changes during a test are rolled back, but the effects of external systems to which the code executed during a test is not. This means that mails are really sent out, data is really sent to external systems, requests to external systems are really logged.

This remains a consideration to not run ATF tests in production instances – unless any external communication is marked as for testing purposes and ALL external systems understand these markers.

Imagine a test simulating a major incident – and part of the major incident management process may be to inform all users of an organization via mail. One wants to know if this process works properly – but one doesn't want to bug tens of thousands of users as part of such a test.

Instance Scan

Instance Scan is an OOTB platform capability that uses so called checks to investigate a defined set of records and based on the configuration or code contained in the check produces a finding.

Findings are produced as the result of a scan – which can be run against an application, an update set, or a single file.

One or more checks can be organized as a scan suite. It is recommended to create a suite that contains all checks relevant for the coding guideline.

Instance Scan is shipped with a limited set of checks – which mainly serve to produce security configuration insights – OOTB there are no checks that assess code or application quality.

A finding is the binary result of a check executed against a single record – by itself a finding does not mean anything. It is neither good nor bad. It depends on its interpretation.

One could use checks to count records that match a defined criteria and – based on thresholds – recommend any kind of action.

In the context of this process, Instance Scan is used as a static code analysis tool and the superset of related checks constitutes the coding guideline. A finding in this context indicates non-compliance to the coding guideline and must be addressed before the affected application can be deployed further downstream.

Instance Scan should not be confused with Heath Scan – which is a ServiceNow service offering using comparable – but not the same – technology.

Heath Scan results however may provide input and inspiration for new checks to be created as part of the coding guideline.

Open-Source Applications

The tools mentioned below are open-source contributions. ServiceNow does not provide any support or warranty. They have been developed based on the insights gained when implementing the described process with clients – and hence might help to shortcut some of the technical implementations required to fully implement the automated QA and deployment procedures described in this document. It is at the customer's discretion to either use these tools as they are – at their own risk – or use them as inspiration on how to implement their own technical solutions.

DevTools

The DevTools application contains hundreds of re-usable functions and classes that extend or encapsulate Glide API or JavaScript features. DevTools has the tools required for cross scope scripting techniques and installation scripts. DevTools also contains an implementation of Dependency and Technical Debt management.

https://github.com/saschawildgrube/servicenow-devtools

Deployer

The Deployer application implements full pipeline automation from installing applications from a source control system to running automated tests.

The Deployer application also allows to verify if an application version that is baselined and stored in a branch in a source control system is ready for deployment by looking into all its dependencies.

https://github.com/saschawildgrube/servicenow-deployer

CodeSanity

The CodeSanity application contains several instance scan checks to validate source and application files.

https://github.com/saschawildgrube/servicenow-codesanity

Runbook

The Runbook application can be used to render the operations manual (runbook) for an instance based on the actual configuration.

https://github.com/saschawildgrube/servicenow-runbook

Agile

The Agile application is a lightweight backlog management system that supports the ideal state flow to support this process. You can manage backlog items in multiple backlogs, components to be worked on and affected personas.

https://github.com/saschawildgrube/servicenow-agile

Installing or Updating open-source applications from GitHub

Follow these steps to install or update applications from open-source repositories hosted in GitHub (github.com). Some steps are only required if the target instance is "air-gapped" and no direct (or mid server relayed) connection to GitHub is possible.

1. If not done already create a GitHub account.

2. Fork the most current state of the open-source application's public repository (you may have to delete an earlier fork).

3. If not done already create an API token with read access to your GitHub account

4. If the target instance is NOT air-gapped

   a. If the application is NOT yet installed on the target instance

      i. Configure the GitHub API access token as a basic auth credential on the target instance

      ii. Navigate to Studio

      iii. Choose "Import from Source Control" and load the application from the forked repository

      iv. Navigate to /sys_repo_config_list.do

      v. Delete the corresponding repository configuration record

      vi. If no repository in the organization's source control system exists

         1. Create the repository in the organization's source control system

      vii. If no API access token exists that allows access to the repository, create one and set it up as a basic auth credential on the target instance

      viii. Navigate to Studio

      ix. Choose "Link to source control" and use the repository url and credentials relating to the organization's source control system. Choose "dev" as the branch name.

      x. Make the "dev" branch the default branch.

      xi. Delete all other branches

   b. If the application is ALREADY installed on the target instance

      i. Navigate to /sys_repo_config_list.do

      ii. Change the repository url and credential to point to the forked repository on GitHub. Change the current branch to "master" (or whatever branch contains the version to be installed).

      iii. Navigate to Studio

      iv. Verify the repository configuration

      v. Select "Apply Remote Changes"

      vi. Navigate to /sys_repo_config_list.do

       vii. Change the repository url and credential to point to the organization's repository in the organization's source control system. Change the current branch to "dev."

5. If the target instance is air-gapped

    a. Create a developer instance on the ServiceNow developer portal

    b. Configure the GitHub API access token as a basic auth credential on the developer instance

    c. Import the open-source application from source control

    d. Publish the application to an update set

    e. Export the Update Set to XML

    f. Transfer the XML file to a location from where it can be uploaded to the target instance

    g. Import the XML Update Set file to the target instance

    h. Preview the loaded Update Set

    i. In case of issues, choose "Accept remote update" in all cases

    j. Commit the Update Set

    k. Ignore all errors

    l. If the application is NOT yet linked to a source control repository in the organization's source control system

       i. If no repository in the organization's source control system exists

          1. Create the repository in the organization's source control system

       ii. If no API access token exists that allows access to the repository, create one and set it up as a basic auth credential on the target instance

       iii. Navigate to Studio

       iv. Choose "Link to source control" and use the repository url and credentials relating to the organization's source control system. Choose "dev" as the branch name.

       v. Make the "dev" branch the default branch.

       vi. Delete all other branches.

6. Baseline the new application version
(see chapter "Application Version Baseline Procedure")

Family Upgrades

Impact on Family Upgrades

Introducing the process described in this document has a huge impact on how ServiceNow platform family upgrades are handled.

Spoiler alert: it is going to be much simpler, faster, less error prone, less laborious, more reliable.

When everything is shipped as an app, and all changes to OOTB or plugin scopes are done through idempotent installation scripts any family upgrade can be treated like a regular deployment.

Of course, a comprehensive test is mandatory. Any new behavior introduced by new or updated components must be tested against the existing applications.

Applications may have to be updated to fix these regressions or to make even better use of new features in the platform.

But the laborious process of reviewing "skipped records", the compilation of Upgrade Plans, the backporting and migration of changes made to OOTB records – all of that is just no longer necessary.

ServiceNow has introduced great improvements to the Family Upgrade process over time – but by applying the proposed process, Family Upgrades are no longer the biggest event of the year, they turn into yet another regular deployment, nothing special, business as usual.

Procedure for Family Upgrades

Follow these steps for a Family Upgrade:

1. Upgrade development instances
    a. Trigger the Family Upgrade
    b. Run all installation scripts
    c. Perform regression test
    d. Adapt applications to modified OOTB or plugin behavior
    e. Re-test
2. Upgrade any downstream instances
    a. Trigger the Family Upgrade
    b. Deploy adapted application versions
    c. Run installation scripts
    d. Perform regression test

Cloning

Do you still need to clone?

The question alone might sound as if from another world. Do you still need to clone after all the guidance provided in this document is applied?

The answer is: maybe not.

If all guidance provided in this document is applied, one can tick off all of the following check points:

- Everything is deployed as an app
- All historical changes to the global scope or plugin scopes have been refactored into applications and installation scripts
- All test data required for development, UAT and other manual test activities can be created automatically on-demand

Once this state is reached – no cloning will ever be necessary.

Getting to a "clean" development or test instance can then be achieved by z-booting the instance, installing all applications, running all installation scripts, automatically creating all necessary test data, and performing a few manual configuration steps to set up instance specific credentials and connections – that's it.

This is however an ideal world scenario. Depending on the age of the instance, the level of customizations, and the technical debt that existed when the proposed process was introduced this scenario may never be reached.

So, for most organizations, cloning will still be part of their routine – although less often than before.

Activities when cloning

Cloning comes with two challenges:

1. Defining the data that should NOT be cloned

2. Protecting unfinished work

The process described in this document does not help with challenge 1. Challenge 2 however, is much easier to master when using only applications that are stored in source-control repositories.

Instead of capturing unfinished work in Update Sets before cloning and re-applying these Update Sets after cloning, the following activities are needed:

Before Cloning (only on development instances)

- Commit all uncommitted changes for all applications

After Cloning

- (Re-)install all applications from source control
- Execute all installation scripts