

Solution Lifecycle Management for

Dynamics 365 for Customer Engagement apps,
Dynamics 365 for Customer Engagement apps (on-premises),
and Common Data Service (CDS) for Apps

VERSION: 1.0

AUTHOR: Phil Hand

COMPANY: Microsoft Corporation

RELEASED: January 2019



Copyright

This document is provided "as-is". Information and views expressed in this document, including URL and other Internet Web site references, may change without notice.

Some examples depicted herein are provided for illustration only and are fictitious. No real association or connection is intended or should be inferred.

This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes.

The videos and eBooks might be in English only. Also, if you click the links, you may be redirected to a U.S. website whose content is in English.

© 2019 Microsoft. All rights reserved.

Microsoft, Active Directory, Azure, Bing, Cortana, Delve, Dynamics 365, Excel, Hyper-V, Internet Explorer, Microsoft Dynamics 365, Microsoft Edge, Microsoft Intune, MSDN, Office 365, OneDrive, OneNote, Outlook, Power BI, PowerPoint, PowerShell, PowerApps, SharePoint, Skype, SQL Server, Visual Studio, Windows, Windows PowerShell, and Windows Server are trademarks of the Microsoft group of companies. All other trademarks are property of their respective owners.

Table of Contents

Figures	v
Tables	vi
Preface	7
Overview	8
Released versions	9
ALM Maturity	10
ALM self-assessment checklist	10
What are the challenges?	11
Import timeouts	11
Solution layering and application behavior	11
Merging configuration.....	11
Collisions	11
Patching solutions	12
Deleting configuration.....	12
Automating ALM processes.....	12
Common Implementation Pitfalls	14
Developing multiple solutions in a single development instance.....	14
Using multiple solution publishers across solutions being developed	14
Lack of good solution version control practices.....	16
Lack of solution segmentation	16
Solution dependency version mismatch between environments.....	16
Manual solution deployment.....	17
Deploying unmanaged solutions to production	18
Solutions & Framework Fundamentals – An introduction or recap	19
What is a Solution?	19
Managed vs Unmanaged Solutions	20
What is Layering?	23
Managed solutions.....	24
Unmanaged solutions	25

Solution Segmentation.....	25
Add all Assets	26
Add Subcomponent.....	26
Include Entity Metadata.....	27
Updating Solution Components.....	29
Formalizing patch management	38
Upgrade Solution versus Stage for Upgrade.....	43
Deleting customizations and components	44
Plugin-types and versioning.....	45
Composing applications.....	47
Defining solution boundaries	47
How many instances are required?.....	49
Solution Lifecycles.....	54
Repeatable and predictable deployment.....	54
Development approach.....	54
Tooling for automation	59
Solution Packager	60
Configuration Migration Tool	62
Package Deployer	63
UI Automation Test Library (Easy Repro)	65
Online Management API.....	65
Process and Automation Maturity.....	71
Where should investment be made?	73
Deployment to downstream environments.....	73
Solution Management Version Control.....	73
Build Management	73
Test Automation	74

Figures

Figure 1 - Pitfalls: developing multiple solutions in a single development instance.....	14
Figure 2 - Pitfalls: using multiple solution publishers	15
Figure 3 - Pitfalls: inconsistent solution versions across environments	17
Figure 4 - Solution Composition	20
Figure 5 - Unmanaged and Managed solution layering interactions.....	24
Figure 6 - Updating components through standard upgrade (incrementing the solution version number).....	30
Figure 7 - In-place patch of components by reuse of existing solution version	33
Figure 8 – Patching components by varying patch solution name	36
Figure 9 - Patching components correctly in V8.0+	39
Figure 10 - Cloning base solution / rolling-up patches	41
Figure 11 - Upgrade solution process flow	43
Figure 12 - Stage for Upgrade process flow.....	45
Figure 13 - Supporting a single version/single solution application	50
Figure 14 - Supporting current and next versions of a single solution application.....	50
Figure 15 – Extending to support a multi-solution application	51
Figure 16 - Supporting current and next versions of a mutli-solution application	52
Figure 17 - Solution propagation through environments.....	54
Figure 18 - High level basic development process	54
Figure 19 - Automating a check-in.....	61
Figure 20 - Automating a build	62
Figure 21 - Creating a package project	63
Figure 22 - Extending the build process to include packaging for deployment	64

Tables

Table 1 - Release names and versions	9
Table 2 - Managed & unmanaged solutions comparison	21
Table 3 - Solution segmentation behaviors.....	25
Table 4 - Factors influencing solution boundary definition.....	48
Table 5 - Instance topologies for development approaches	55
Table 6 - Approaches for resetting to a known state.....	66
Table 7 - Process and Automation Maturity Matrix.....	71

Preface

Application Lifecycle Management (ALM) is about how you can effectively continue to ship a solution to customers while adapting to both internal and external changes. This content builds upon a [ALM for Microsoft Dynamics CRM 2011](#) whitepaper using insights gained from implementations by enterprise customers, partners and ISVs. Since the original whitepaper was released, the [Common Data Service \(CDS\) for Apps](#) platform has become the centerpiece for the [Microsoft Business Application Platform](#) and this content will cover those changes as well as new considerations introduced with cloud transformations.

Despite all the changes, most of the steps and approaches have not fundamentally changed. Also, the capabilities of the solution framework have evolved to help make some things simpler and more efficient. This content focuses on Online environments, but most techniques and approaches remain relevant for on-premises and Azure hosted deployments.

You will learn:

- Why it is important to utilize the enhancements to the solutions framework
- What approaches can be taken to leverage those capabilities.
- How to incorporate within an ALM process for a successful implementation

If you have been implementing solutions for several years, you may perceive certain challenges related to solutions which you may have encountered with early versions of the solution framework. These challenges may have caused you to hesitate in adopting enhancements. This content will address those concerns and the workaround for them will be used as examples for common challenges and to describe good practices with the current platform version.

Overview

Agile techniques are now common place within Microsoft Dynamics 365 for Customer Engagement implementations and a DevOps mindset is influencing the approaches organizations adopt to maintain healthy, sustainable Line of Business (LoB) applications. The net result enables organizations to gain competitive advantage by implementing processes that enable rapid adoption of enhancements to the Dynamics 365 for Customer Engagement platform and standalone [model-driven apps](#) in PowerApps.

This has also been true for Microsoft, and the Dynamics 365 for Customer Engagement platform is rapidly evolving. The rate of change and need for a seamless update experience with minimal impact requires clear separation of the application(s) from the platform. This provides the ability to update the underlying platform independently of the applications. The side effect of this has required application behavior that previously existed within the platform to be extracted and made solution aware, which has also required further enhancements and evolution of the solutions framework.

The Dynamics 365 for Customer Engagement app modules and standalone [model-driven apps](#) in PowerApps are now delivered and installed as solutions in the same way that any organization delivers apps to their business. The enhancements to the solutions framework required to support the application/platform separation equally benefit customers, partners and ISVs implementing for Dynamics 365 for Customer Engagement apps or the [Common Data Service \(CDS\) for Apps](#).

When adopted, these enhancements simplify the processes and improve efficiency across all phases of the Application Lifecycle.

From an online perspective, it may be easy to overlook changes that are perceived to be minor – such as the ability to patch and upgrade solutions or to be able to define composition at a sub-component level.

For on-premises or IaaS hosted deployments, sometimes the capabilities are not realized due to longer upgrade cycles.

In each situation, simplifying application lifecycle management and reducing time to value will only be fully realized by leveraging the enhancements to the solutions framework.

Released versions

There have been many releases of Microsoft Dynamics CRM, including a renaming to Microsoft Dynamics 365 for Customer Engagement apps. For readability and comprehension, the remainder of this white paper will use release version numbers to avoid any ambiguity.

The table below serves as a reference

Release Name	Release Version
Microsoft Dynamics CRM 2011	5.0
Microsoft Dynamics CRM 2013	6.0
Microsoft Dynamics CRM 2013 SP1	6.1
Microsoft Dynamics CRM 2015	7.0
Microsoft Dynamics CRM 2015 (SP1/Update 1)	7.1
Microsoft Dynamics CRM 2016	8.0
Microsoft Dynamics CRM 2016 (SP1/Update 1)	8.1
Microsoft Dynamics 365	8.2
Microsoft Dynamics 365 for Customer Engagement apps	9.0
Microsoft Dynamics 365 for Customer Engagement apps – April 2018 Update	9.0.2
Microsoft Dynamics 365 for Customer Engagement apps – October 2018 Update	9.1

Table 1 - Release names and versions

ALM Maturity

Before reading this whitepaper, review the checklist below which details the expected level of proficiency for solution implementation.

If the answer is “No” to 2 or more of the checklist items, read the entire whitepaper.

ALM self-assessment checklist

☐ **Solution Management**

- ☐ Solution boundaries are defined as they relate to production – not organized just to simplify development
- ☐ There is at least one discrete development instance per solution being developed and serviced
- ☐ Where a solution being developed has dependencies on one or more solutions, the dependencies are satisfied through always importing those solutions as managed solutions into the discrete development instance for the solution being developed
- ☐ Where multiple solutions are developed for deployment to a single production environment, the same solution publisher and prefix are used
- ☐ The term “Solution Segmentation” is understood
- ☐ Solution Segmentation is practiced consistently
- ☐ The term “Patch Solution” is understood
- ☐ Use of patch solutions is practiced consistently for developing solution fixes
- ☐ Solution.zip files and their contents are never manually edited

☐ **Version Control & Build Management**

- ☐ Solutions are unpacked using the Solution Packager and version controlled in a repository such as Team Foundation Server or GitHub
- ☐ Solutions are built through an automated process that invokes the Solution Packager to repack the solution into a Solution.zip file
- ☐ Packages are built through an automated process that combines the required solutions and configuration data into a PackageDeployer.zip file

☐ **Test Management**

- ☐ Each test case has traceability back to requirement
- ☐ Test cases are automated

☐ **Deployment Management**

- ☐ Only managed solutions are deployed to environments downstream of development
- ☐ All solutions are deployed via the Package Deployer
- ☐ No unmanaged changes are made directly to environments downstream of development
- ☐ The use of patch solutions is practiced consistently for deploying solution fixes

What are the challenges?

The solutions framework capabilities have evolved significantly from their inception in Dynamics CRM 2011 (version 5.0). Common challenges that continue to be observed within enterprise deployments tend to be related to not fully utilizing the platform enhancements introduced by later versions.

Import timeouts

In the version 5.0 era, implementation teams often chose to use an unmanaged solution approach for deployment. This was like the approach taken for earlier platform versions and frequently meant that the solution imported contained all configuration changes rather than delta differences. The solution framework was also not as optimized as it is today, and in some instances the combination of those factors led to import timeouts.

To overcome the timeout challenge, implementation teams often tried to segment their solutions to reduce the likelihood of being impacted by timeouts. Various implementation approaches such as segmenting by component type (e.g. security role, plug-in, entity schema) were observed which often created their own challenges such as requiring a solution to be imported multiple times.

Solution layering and application behavior

A frequently perceived challenge is the ability to predictably control what application behavior the user is ultimately presented with. This is typically caused through the combined use of managed solutions and unmanaged customizations within the system. It can also be caused through implementation-specific approaches to patch managed solutions and the import order of solutions.

Merging configuration

Merging applies to Forms, SiteMap and RibbonDiffs. The behavior occurs when multiple solution layers exist for these assets. Each managed solution layer merges with other layers and the system layers to present application behavior to the user.

Often within Enterprise scenarios, multiple Dynamics CRM or Dynamics 365 for Customer Engagement instances are used to enable team member or workstream isolation. These require merging of changes for downstream environments which can result in challenges stemming from the solution import order that impact application behavior.

Collisions

Prior to version 8.0, when one or more team members are customizing the same entities within a single instance, it was not possible to export team member A's entity changes without also exporting team member B's changes. Version 8.0 introduced [Solution Segmentation](#), providing tighter control on the components added to a solution by enabling discrete sub-components of an entity to be added. Collisions can still occur though which is typically seen as a challenge in larger Enterprise scenarios that haven't adequately isolated team members or workstreams, or where control over the (sub)components added to a solution has not been strict enough.

Patching solutions

Prior to version 8.0, patching a managed solution was a relatively straightforward process. However, it is common for the full capability of the platform to be overlooked and sometimes misunderstood. Patching capabilities have been further improved and formalized from version 8.0 onwards.

A patching approach that is often observed is to create a new solution that takes the name of the solution to be patched, suffixed by the patch number. As the solution name has changed, multiple layers are created (one per patch solution on import) and the component reference count is incremented once for each layer containing the component.

The following challenges have been observed:

- Time to import - Often a simple solution upgrade approach is adopted (and mistakenly referred to as a “patch”) rather than taking advantage of patching capabilities. This means each “patch” solution unnecessarily contains a full copy of the original solution plus the amendment. Taking this approach impacts time to import unnecessarily when unchanged components are included in the patch and are assessed for change
- Layering order of patches – can be difficult to assess and will likely be based on order of import, which could be susceptible to error in downstream environments unless automated to control import order
- Number of patch solutions – where multiple patches are applied to a solution in this manner, it becomes increasingly more effort to remove a component from the system.

Deleting configuration

Deleting a component has always been possible with a manual holding solution technique. This approach is somewhat cumbersome and increases in complexity if the approach to patching solutions has been one where the solution name is changed for each patch.

Implementors are generally aware of how to delete components in theory but the challenge is often to identify what other solutions are preventing deletion due to having a dependency on the component that is being removed. This can be a time consuming and resource intensive process that is often manually executed.

Automating ALM processes

A good practice approach when commencing on a new work item or new test cycle is to first reset the environment to a known state. Traditionally, with on-premises environments, this was achieved by either restoring the DB from backup or resetting the Virtual Machine, followed by deploying the relevant build held within version control. The process is automated to ensure it is executed consistently.

Adopting the same approach within an online environment has been a challenge until more recently. Prior to the Version 9 timeframe, key activities to automate such as resetting an instance or restoring from a known backup were not possible. The [Online Management API](#) now provides REST-based operations to create and manage instances. Currently, there is no specific “Reset Instance” operation but the equivalent can be achieved through a Delete Instance operation followed by a Create Instance operation. Further details are provided in [Online Management API](#) section later in this document.



Important: If you observe any of the above challenges within your implementation – read the entire whitepaper

All the challenges described above can be reduced or eliminated through appropriate use of the investments that have been made within the platform and solutions framework.

The following sections of the document will highlight good practice approaches on the current platform version that will help to avoid the challenges that have been commonly observed.

Common Implementation Pitfalls

Developing multiple solutions in a single development instance

This approach is responsible for a large proportion of implementation issues. All changes exist in the unmanaged layer. The notion of being grouped into a solution only applies when the solution container that is used to logically group a set of unmanaged changes is exported.

There is no isolation between the unmanaged components and therefore it is easy for one or more components to be changed and included in more than one solution container. When deploying to downstream environments, change in application behavior may not be predictable because it will be based on the order of solution import or the layering order for solutions present in the system.

It is also possible to inadvertently create cyclic dependencies that prevent solutions importing. Consider the following unmanaged changes logically grouped into solution containers:

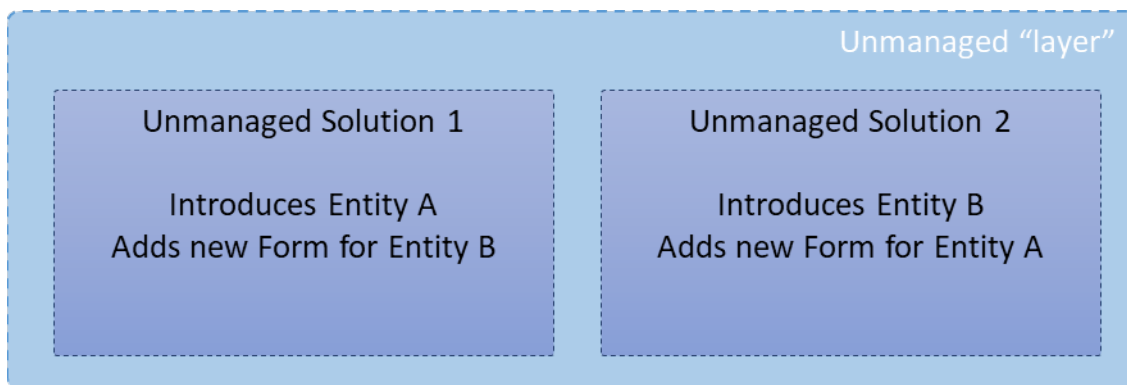


Figure 1 - Pitfalls: developing multiple solutions in a single development instance

Assuming segmentation is used, Solution 1 creates a new form for Entity B and Solution 2 creates a new form for Entity A. The solutions are exported as managed solutions and exclude required components.

Solution 1 can't be imported because Entity B does not exist within the target environment. Solution 2 can't be imported because Entity A does not exist within the target environment.

Because a single development instance was used to develop both solutions, each solution has been polluted with unmanaged change from the other. This would require dependencies to be cleaned up and isolated which in practice may not be straightforward with more complex solutions. The alternative would be to move all unmanaged change into a single solution which is likely to be impractical given the perceived need for multiple solutions in the first place.

Using multiple solution publishers across solutions being developed

Often solution boundaries are defined based on functional need, for example a sales solution, a service solution both taking a dependency on a common core solution. Sometimes there is a need

for components to be moved between these solutions for example, a component in the Service solution is now also required by the Sales solution. The approach would be to move this component to the Common Core solution. If different solution publishers have been used between the solutions, this is not possible without deleting and recreating the component. Consider the following structure:

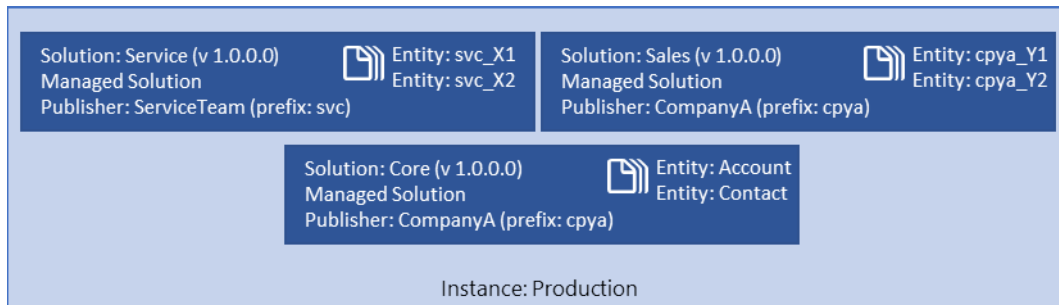


Figure 2 - Pitfalls: using multiple solution publishers

The Core solution extends the Account and Contact entities. Both the Sales and Service solutions take dependencies on these components.

The Service solution has a unique publisher, therefore assuming component "svc_X2" was determined to be common, it is not possible to move it directly into the Core solution as the Service solution maintains a reference to it. It is not possible to create an identical component directly in the Core solution as it would flag a conflict on import due to the fact it was introduced by a different publisher.

The only approach remaining is to introduce a new component within the core solution, using a holding solution technique to delete the original component from the service solution and "optionally" conduct a data migration between the two components.

Note: it is likely there would be additional work to remove any relationships to the entity being removed and to create new relationships to the entity being created which would make this a very complex process to maintain data integrity within a live production environment.

The Sales solution uses the same publisher as the Core solution. This enables "cpya_Y2" to easily be moved as an unmanaged component to the core solution and be removed from the Sales solution within the development environment. The solutions can then be reimported into production and the component will continue to exist given the solution publisher maintains a reference count of 1 or greater to the component throughout the process.

Lack of good solution version control practices

Knowing who changed what when is critical for all software development including model-driven app solutions. Granular version control enables faster identification of regressions and changes to behavior.

Storing versions of the exported solution.zip file in source control is analogous to storing a compile binary in source control. It is not possible to easily identify what has changed, when it changed and who changed it.

Just as the source code for a .NET binary should be stored in source control, the unpacked XML fragments of the exported solution.zip file should be stored in source control. This step is still found to be lacking in a number of customer implementations.

Lack of solution segmentation

Using solution segmentation minimizes the likelihood of collisions when developing and merging solutions. It also avoids the introduction of unnecessary dependencies.

Conversely, for implementation that do not practice solution segmentation, the likelihood of collisions is higher, leading to additional effort to remediate and harmonize changes. Additional and unnecessary dependencies introduced can lead to issues when deploying to downstream environments or when deleting components from downstream environments.

Solution dependency version mismatch between environments

This issue is most likely to occur when solution segmentation is not practiced. A solution being developed may take a dependency on a component within a managed solution that was introduced with a specific version of that managed solution.

If the target environment has a lower version of the managed solution, the solution being developed will not be able to be imported as its dependencies will not be satisfied (given the component it is dependent upon does not exist within the target environment).

Occurrences of this issue are most common where there is a dependency on a first party solution (Microsoft app). These apps are serviced by applying the update to a target instance via the Instance Picker.

INSTANCES
UPDATES
SERVICE HEALTH
BACKUP & RESTORE
APPLICATIONS

Manage your solutions

Manage your solutions

Select a preferred solution to manage on selected instance: Dev1

SOLUTION NAME	VERSION	AVAILABLE UNTIL	STATUS
Company News Timeline	1.0.3	1/1/2050	Upgrade available
Crm Hub	1.0.20170908.55	1/1/2050	Not installed
Customer Service Hub	9.0.5.61	1/1/2050	Installed
Data Export Service for Dynamic...	1.0.0.0	1/1/2021	Installed
Dynamics 365 Customer Service ...	2.0.0.4	1/1/2050	Installed
Dynamics 365 for Talent Anchor	6.100.1	1/1/2021	Not installed
Dynamics 365 Portals - Base Por...	9.0.6.1	1/1/2050	Not installed
Dynamics 365 Portals - Commun...	9.0.6.1	1/1/2050	Not installed
Dynamics 365 Portals - Custom ...	9.0.6.1	1/1/2050	Not installed
Dynamics 365 Portals - Custome...	9.0.6.1	1/1/2050	Not installed
Dynamics 365 Portals - Employee...	9.0.6.1	1/1/2050	Not installed
Dynamics 365 Portals - Partner P...	9.0.6.1	1/1/2050	Not installed
Dynamics 365 Portals - Partner P...	8.4.0.275	1/1/2050	Not installed
Dynamics 365 Sales Application	9.0.0.4	1/1/2050	Upgrade available

Dynamics 365 Sales ...

UPGRADE
Automate sales processes and monitor performance with this customizable relationship management app

Created by: Microsoft Dynamics

New version: 9.0.1810.5012

[Learn more](#)

Figure 3 - Pitfalls: inconsistent solution versions across environments

If the solution being developed does not use solution segmentation, there is more chance for additional dependencies to be carried in the payload of the solution. This means that if the target environment does not have an up to date version of the managed solution on which the solution being imported is dependent, there is more likelihood there will be a dependency that cannot be satisfied. This will result in a solution import failure.

Manual solution deployment

The benefits of automated software deployment are clear. It enables a repeatable, predictable deployment that anyone with appropriate privileges should be able to execute. This is standard good practice throughout the software industry.

The same good practice applies to solution and package deployment however it remains commonplace for organizations to deploy solutions manually. Manual deployment requires a higher knowledge of the target environment and introduces the human error factor.

Deploying unmanaged solutions to production

Deployment of unmanaged solutions is not true deployment. It is the placement of unmanaged component customizations into the target environment. This is the same as making that change directly within the target environment and is analogous to directly updating source code.

Environment governance is important to most if not all organizations.

- Would you allow an authorized user to directly change source code in your production customer facing web site?
- If not, you should not allow an authorized user to directly change source code in your production Line of Business application.



Important: If you observe any of the above pitfalls within your implementation – read the entire whitepaper

Solutions & Framework Fundamentals – An introduction or recap

A good level of understanding of the solutions framework capabilities is required to appreciate the approaches that can be used to solve the needs for an implementation. These fundamentals, together with more advanced concepts are documented within the [Package and distribute extensions using solutions](#) section of our published documentation.

This section is intended to provide Dynamics CRM veterans with a quick re-cap on the evolution of the platform capabilities while also allowing those that are new to the platform to quickly understand current capabilities.

What is a Solution?

A solution is a container for components of the application. Solutions act as the unit of transport to enable applications to be moved between environments.

Solutions can be used to ensure all new components have a standard naming prefix and are the mechanism to author, package, maintain and distribute functional configuration and custom code that extends the behavior of Microsoft Dynamics 365 for Customer Engagement apps.

To enable clear application and platform separation with Version 9.0, additional components have been made solution aware which has the benefit of simplifying the propagation of these components between development, test and production environments.

A solution may comprise of many components which can be categorized broadly into four areas as shown on the diagram below:



Figure 4 - Solution Composition

Managed vs Unmanaged Solutions

The introduction of solutions in Version 5.0 sparked considerable debate within the community in terms of whether to use managed or unmanaged solutions. Several qualities of managed solutions had not fully matured which often led to two misuses of unmanaged solutions in preference.

- 1) **Familiarity:** Implementors using unmanaged solutions through all environments owing to the fact they "act" in a similar way to customizing in Version 4.0 and below.
- 2) **Complexity:** Managed solutions were often perceived to cause complexity. Unintended dependency issues sometimes made them difficult to remove and the order of import could influence the application behavior presented to the user if not reliably controlled.



Important: From Version 6.0 onwards, Microsoft have reinforced recommended good practice and in particular, for organizations that require the ability to govern, remove, patch and upgrade production deployments:

Unmanaged Solutions: Analogous to application source code and to be used for **development purposes only** – to act as a container for unmanaged customizations and to transport between development instances.

Managed Solutions: Analogous to compiled application binaries – to be deployed (imported) into **any instances downstream of development environments**. Also, to be deployed within development instances where used as a common library providing base capability that will be extended for a discrete purpose or implementation.

There are important differences between unmanaged and managed solutions which requires good understanding of how layers are created and interact with one another. This is documented in detail within the [Introduction to solutions](#) section of the published documentation. The differences between the two types of solutions are summarized in the table below.

Table 2 - Managed & unmanaged solutions comparison

Unmanaged Solution	Managed Solution
Acts as a container of customizations that exist on top of the system and other managed solutions	A "version complete" unmanaged solution that has been exported as managed
Unmanaged solutions are "Open"	Managed solutions are "Closed"
<ul style="list-style-type: none">• They allow direct customization	<ul style="list-style-type: none">• The identity of the solution

<p>to components</p> <ul style="list-style-type: none"> • They hold references to customization and components but do not own them • They do not delete or remove components if the solution is deleted 	<p>cannot be modified</p> <ul style="list-style-type: none"> • Components cannot be added or removed • They exist within their own discrete layer • They can be upgraded and serviced as discrete layers enabling ISV's or multiple departments to provide multiple applications to a shared environment • They merge with other layers and the system layers to present application behavior to the user • They can be uninstalled (deleted) and they do delete components
---	--

	that they directly own
--	---------------------------

What is Layering?

Layering occurs on import of solutions when a specific component is affected by change within one or more solution. Layers describe the dependency chain of a component from the root solution introducing it, through each solution that extends or changes the component's behavior. Layers are created through extension of an existing component (taking a dependency on it) or through creation of a new component or version of a solution. Examples include:

- Extending a managed Common Data Model (CDM) component – which takes a dependency on the component
- Extending a managed component from a first-party app (Microsoft App) – which takes a dependency on the component from the solution that represents the app
- Extending a managed component from an ISV solution or custom solution/app
- Introducing a new custom component via a custom managed solution – which creates the component and consequently the root layer for that component.
- Amending properties of an existing component within a different managed solution

Layering is important because it enables the application behavior to be controlled, updated and patched.

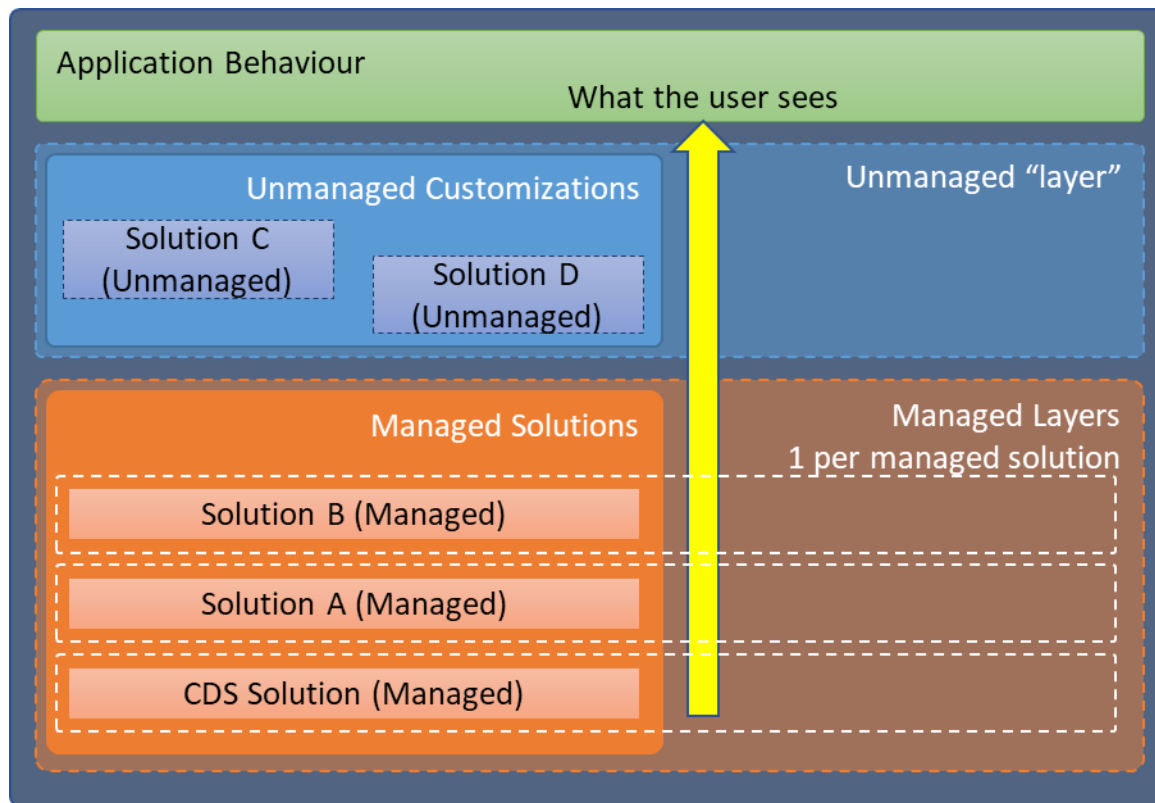


Figure 5 - Unmanaged and Managed solution layering interactions



Important:

Layers should be considered on a per-component basis. Although typically drawn as above to convey the effect of a solution on another solution, this is always at a component or sub-component level.

For each managed component imported into the environment, there is no unmanaged layer when it is introduced. Unmanaged customizations only exist when a customizer amends the component.

Managed solutions

Managed solutions create layers on a per-component basis, they can also extend and take dependencies upon other managed solutions. Both the import order and the dependencies between managed solutions will influence the layering order of the managed solutions. This is particularly important to appreciate when two or more managed solutions with no interdependencies contain a reference to a common component. In such circumstances, the import order may change the layering order, potentially leading to unintended application behavior.

Unmanaged solutions

Unmanaged solutions do not create any layers. They are simply containers to logically group unmanaged component customizations and when imported, will add the contained unmanaged component customizations to the unmanaged layer. Importantly, unmanaged component customizations override any equivalent managed component customization introduced by managed solutions that layer beneath them.

Component customizations cannot be deleted through deletion of an unmanaged solution, they can continue to mask further updates to those components that are imported through an updated version of an existing managed solution. It is possible to import a managed solution and specify that unmanaged customizations should be overridden. This process essentially updates the managed solution layer and applies the same updated component customizations to the existing unmanaged customizations.

Why is this behavior important to understand?

Depending upon the existing unmanaged component customizations within the system, there is potential to change application behavior and impact what the user sees.

Once unmanaged component customizations have been introduced to the system, there is no method to delete them. Currently, the only method to change their behavior is to make a compensating change – either through a direct update to the unmanaged component customization, or through overriding unmanaged customizations when importing an updated version of an existing managed solution.

Solution Segmentation

Solution segmentation was introduced as part of Version 8.0 and provides more granular control over the assets that are included within exported solutions. There are 3 types of behavior that are used depending upon whether solution segmentation is applied.

Table 3 - Solution segmentation behaviors

	Include Entity Metadata	Exclude Entity Metadata
Add Entity to solution	Behavior = 1	Behavior = 2
Add subcomponent	Behavior = 1	Behavior = 2
Add All Assets (forces "Include Entity Metadata" in addition to all subcomponents)	Behavior = 0	N/A

The behavior is held as an attribute of the RootComponent XML element for the entity within the solution.xml file. This can be verified after solution export to confirm the appropriate approach to solution segmentation has been configured.

Add all Assets

Selecting Add all Assets when an entity is added to a solution and then exporting that solution will result in the entity and all its assets being exported in the solution. This includes attributes, forms, views, visualizations, relationships, business rules and any other assets that are packaged with the entity. It also includes the entity metadata. This has the consequence of exporting all assets that have been amended and all those that have not been amended for the entity. This process can result in the unintended consequence of including dependencies and or modifying unintended assets on the target environment.

The snippet below shows an XML fragment from the solution.xml for a solution containing the account entity with "Add all Assets"

```
<RootComponents>
  <RootComponent type="1" schemaName="account" behavior="0" />
  <RootComponent type="60" id="{15915835-b87c-49ec-9ffc-bfac3ac44ef1}" behavior="0" />
</RootComponents>
```

The customizations.xml includes the entity, all the entity's subcomponents and the metadata describing the entity.

When to use this option

- Creation of a new entity that is added to the solution
- Deployment of the entity to a new environment or to one that does not have a previous version of the entity installed

Add Subcomponent

By not selecting Add all Assets, each asset that is required from the entity can and must be selected specifically. This provides granular control over the assets contained within the solution when exported and deployed. This approach prevents the unintended consequence of including unnecessary dependencies and or modifying unintended assets on the target environment.

When adding an asset to the solution, a dependency check determines if there are any dependencies that may also require inclusion. If these are already present in the target environment and remain unchanged, the solution developer can choose to disregard them.

The snippet below shows an XML fragment from the solution.xml for a solution containing the account entity (with Include entity metadata) and a specific sub-component.

```
<RootComponents>
  <RootComponent type="1" schemaName="account" behavior="1" />
</RootComponents>
```

The behavior is observed to be 2 if the entity metadata is excluded

```
<RootComponents>
  <RootComponent type="1" schemaName="account" behavior="2" />
</RootComponents>
```

When to use this option

- When producing a patch for an already deployed solution
- Enhancing or amending a solution that is already deployed where a subset of assets needs to be updated.

Include Entity Metadata

When a managed entity is added to the solution, the metadata for the entity is included by default. This can be deselected as required.

Entity metadata can later be included (or removed) for by selecting the entity within the solution, selecting "Add Subcomponents" and checking or unchecking the "Include Entity Metadata" checkbox.

Creating a new unmanaged entity and adding it to the solution, forces Add all Assets and Include entity metadata to be selected.

Including entity metadata adds content to the customizations.xml on export of the solution. This metadata will be important to include if there have been any changes that need to be applied to target environment when installed (such as enable audit or enable mobile for the entity). Conversely, if downstream environments follow strict configuration control, it is prudent to avoid inclusion of entity metadata unless a specific change is required. This can help to prevent unintended changes in behavior (e.g. audit unintentionally becoming disabled for the entity)

When to use this option

- Deploying a new entity (enforced)
- Amending an entity's behavior – for example availability on mobile

The snippet below shows an XML fragment from the customizations.xml for a solution containing the account entity, and the metadata for the account entity.

```
<EntityInfo>
  <entity Name="Account">
    <LocalizedNames>
      <LocalizedName description="Account" languagecode="1033" />
    </LocalizedNames>
    <LocalizedCollectionNames>
      <LocalizedCollectionName description="Accounts" languagecode="1033" />
    </LocalizedCollectionNames>
    <Descriptions>
```

```

        <Description description="Updated Description&#xA;Business that represents a
customer or potential customer. The company that is billed in business transactions."
languagecode="1033" />
    </Descriptions>
    <attributes />
    <EntitySetName>accounts</EntitySetName>
    <IsDuplicateCheckSupported>1</IsDuplicateCheckSupported>
    <IsBusinessProcessEnabled>1</IsBusinessProcessEnabled>
    <IsRequiredOffline>0</IsRequiredOffline>
    <IsInteractionCentricEnabled>1</IsInteractionCentricEnabled>
    <IsCollaboration>0</IsCollaboration>
    <AutoRouteToOwnerQueue>0</AutoRouteToOwnerQueue>
    <IsConnectionsEnabled>1</IsConnectionsEnabled>
    <EntityColor>#794300</EntityColor>
    <IsDocumentManagementEnabled>1</IsDocumentManagementEnabled>
    <IsOneNoteIntegrationEnabled>1</IsOneNoteIntegrationEnabled>
    <IsKnowledgeManagementEnabled>0</IsKnowledgeManagementEnabled>
    <IsSLAEnabled>0</IsSLAEnabled>
    <IsDocumentRecommendationsEnabled>0</IsDocumentRecommendationsEnabled>
    <IsBPFEntity>0</IsBPFEntity>
    <OwnershipTypeMask>UserOwned</OwnershipTypeMask>
    <EntityMask>ActivityPointer</EntityMask>
    <IsAuditEnabled>0</IsAuditEnabled>
    <IsRetrieveAuditEnabled>0</IsRetrieveAuditEnabled>
    <IsRetrieveMultipleAuditEnabled>0</IsRetrieveMultipleAuditEnabled>
    <IsActivity>0</IsActivity>
    <ActivityTypeMask></ActivityTypeMask>
    <IsActivityParty>1</IsActivityParty>
    <IsReplicated>1</IsReplicated>
    <IsReplicationUserFiltered>1</IsReplicationUserFiltered>
    <IsMailMergeEnabled>1</IsMailMergeEnabled>
    <IsVisibleInMobile>1</IsVisibleInMobile>
    <IsVisibleInMobileClient>1</IsVisibleInMobileClient>
    <IsReadOnlyInMobileClient>0</IsReadOnlyInMobileClient>
    <IsOfflineInMobileClient>1</IsOfflineInMobileClient>
    <DaysSinceRecordLastModified>0</DaysSinceRecordLastModified>
    <MobileOfflineFilters>&lt;fetch version="1.0" output-format="xml-platform"
mapping="logical" distinct="false"&gt;&lt;entity name="account"&gt;&lt;filter
type="and"&gt;&lt;condition attribute="modifiedon" operator="last-x-days"
value="10"/&gt;&lt;/filter&gt;&lt;/entity&gt;&lt;/fetch&gt;</MobileOfflineFilters>
    <IsMapiGridEnabled>1</IsMapiGridEnabled>
    <IsReadingPaneEnabled>1</IsReadingPaneEnabled>
    <IsQuickCreateEnabled>1</IsQuickCreateEnabled>
    <SyncToExternalSearchIndex>1</SyncToExternalSearchIndex>
    <IntroducedVersion>5.0.0.0</IntroducedVersion>
    <EnforceStateTransitions>0</EnforceStateTransitions>
    <EntityHelpUrlEnabled>0</EntityHelpUrlEnabled>
    <EntityHelpUrl></EntityHelpUrl>
    <ChangeTrackingEnabled>1</ChangeTrackingEnabled>
    <IsEnabledForExternalChannels>1</IsEnabledForExternalChannels>
    <HasRelatedNotes>True</HasRelatedNotes>
    <HasRelatedActivities>True</HasRelatedActivities>
</entity>
</EntityInfo>

```

The impact of not adopting a minimal solution approach through solution segmentation is clearly evidenced through inspection of the solution.xml and customizations.xml files produced for a

solution that has applied the “Add all Assets” approach. The introduction of unnecessary and unused dependencies should be avoided.



Important: Solution segmentation can reduce collisions caused by multiple team members working on assets of a component such as an entity.

Additionally, solution segmentation (only adding assets that are affected to the solution) helps to reduce the size and complexity of the solution imported resulting in reduced import times.

Updating Solution Components

Since version 5.0 there have been three common approaches observed for updating solutions. These have different advantages and disadvantages but of most importance is how the order of solution import can impact application behavior.

Approach 1 – Standard Upgrade (Increment solution version number)

This approach retains the existing unmanaged solution container including the solution publisher, solution name and solution version).

Optionally for import efficiency, any unchanged components are stripped from the solution container, leaving only the components that have been amended. The solution version number is incremented, and the solution is exported as a managed solution.

When imported, the solution creates a new layer directly above the previous layer for the solution.

This technique simplifies identification that an update has been applied but it can be misleading. It is not possible to revert to the previous version of the solution. Therefore, to remove the component changes in the upgrade requires a compensating change.

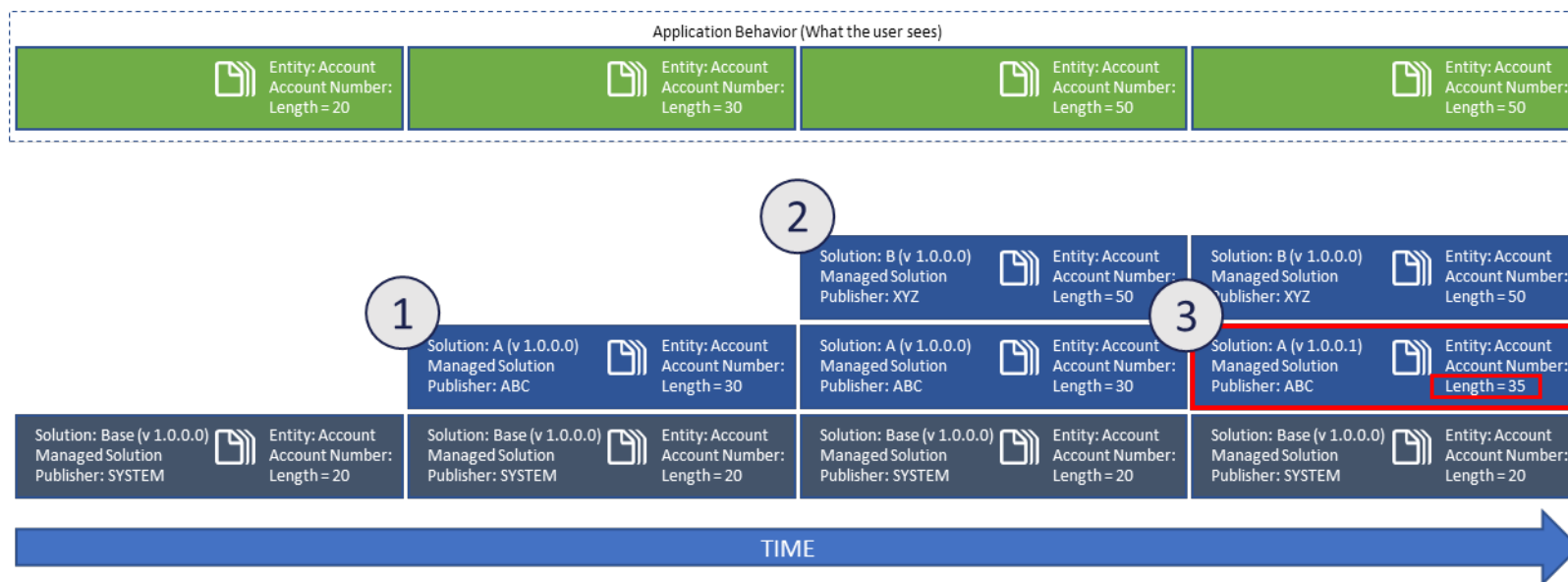


Figure 6 - Updating components through standard upgrade (incrementing the solution version number)

- 1) The Account Number field length is changed to 30. Solution A version 1.0.0.0 is imported. Application behavior reflects the change to Account Number field length.
- 2) Solution B amends the Account Number field length to 50. Solution B version 1.0.0.0 is imported and layers above Solution A based on the import time stamp. Application behavior reflects the change to Account Number field length.
- 3) The Account number field length is changed to 35 within Solution A. The solution version number is incremented to 1.0.0.1 and imported. Solution A version 1.0.0.0 creates a new layer directly above the previous layer for the solution – effectively replacing it. Application behavior does not reflect the change to Account Number field length as Solution B is layered above and masks the change.

The illustration above highlights the change made when the final import is introduced to the appropriate layer within the managed stack. This does not affect application behavior given that Solution B remains on top. It does mean that if Solution B required removal at a later

point in time, Solution A would become the top most layer and would present the patched version as application behavior (i.e. Account Number Length would change to 35).

It is not possible to revert to the version 1.0.0.0 state of Solution A and therefore deletion of changes introduced with version 1.0.0.1 is more complex, requiring a compensating change. Predictable application behavior is achieved because solution layering using this technique remains consistent.

Approach 2 – In-place Patch (re-use solution container and version)

This approach retains the existing unmanaged solution container including the solution publisher, solution name and solution version).

Optionally for import efficiency, any unchanged components are stripped from the solution container, leaving only the components that have been amended for the patch. The solution is exported as a managed solution.

When imported, the solution updates components in the existing layer for the solution.

The weakness of this approach is the lack of ability to identify that a patch has been applied. This is typically overcome by adding a note of the change within the description field of the solution. Additionally, to remove the component changes in the upgrade requires a compensating change.

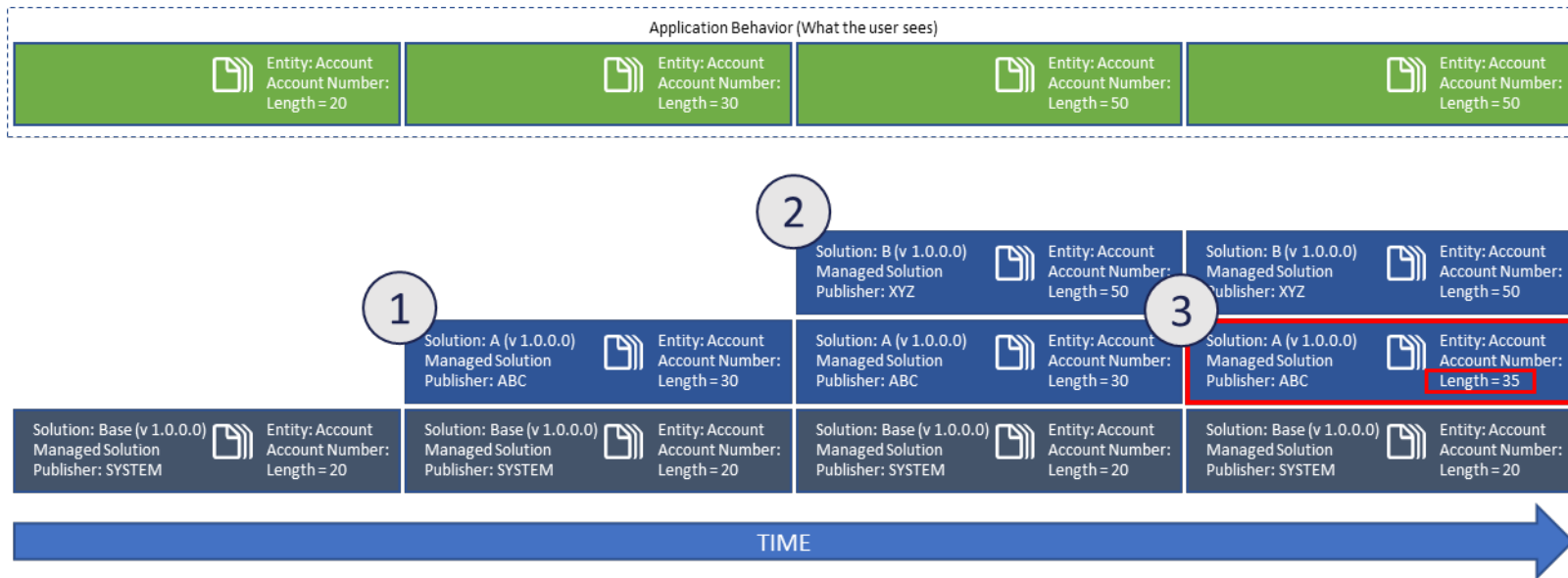


Figure 7 - In-place patch of components by reuse of existing solution version

- 1) The Account Number field length is changed to 30. Solution A version 1.0.0.0 is imported. Application behavior reflects the change to Account Number field length.
- 2) Solution B amends the Account Number field length to 50. Solution B version 1.0.0.0 is imported and layers above Solution A based on the import time stamp. Application behavior reflects the change to Account Number field length.
- 3) The Account number field length is changed to 35 within Solution A. The solution version number 1.0.0.0 is retained. The layer created by Solution A version 1.0.0.0 is directly replaced. Application behavior does not reflect the change to Account Number field length as Solution B is layered above and masks the change.

In-place solution upgrade by updating the components within the existing solution container produces an outcome equal to that of approach 1 (where the existing solution version number was incremented).

The illustration above highlights the change made when the final import is introduced to the appropriate layer within the managed stack. This does not affect application behavior given that Solution B remains on top. It does mean that if Solution B required removal at a later

point in time, Solution A would become the top most layer and would present the patched version as application behavior (i.e. Account Number Length would change to 35).

It is not possible to revert to the previous version 1.0.0.0 state of Solution A as it has been directly overwritten and therefore identification and deletion of changes introduced with the updated version 1.0.0.0 solution is more complex, requiring a compensating change. Predictable application behavior is achieved because solution layering using this technique remains consistent.

Approach 3 – Patch by Amending solution name

Similar in nature to the previous approaches, this technique requires only the components that have been amended for the patch. The solution name is amended to denote the patch number prior to exporting as a managed solution. Sometimes, the version number is set higher than that of the solution it is “patching” – however, it is the change in solution name that affects the layering as the patch is effectively a new solution being introduced.

When imported, the solution follows the standard managed solution import process and therefore creates a new layer at the top of the managed stack.

This approach is an anti-pattern as it can create deviations in application behavior between development, test and production environments due to the order of solution import applied. The approach requires that all solutions be installed in the correct order for layering and application to be correct. If they are installed in a different order a different result will occur. This makes the patch mechanism appear to be less reliable.

Figure 8 below, highlights application behavior as a patch solution is introduced into the environment using this approach.

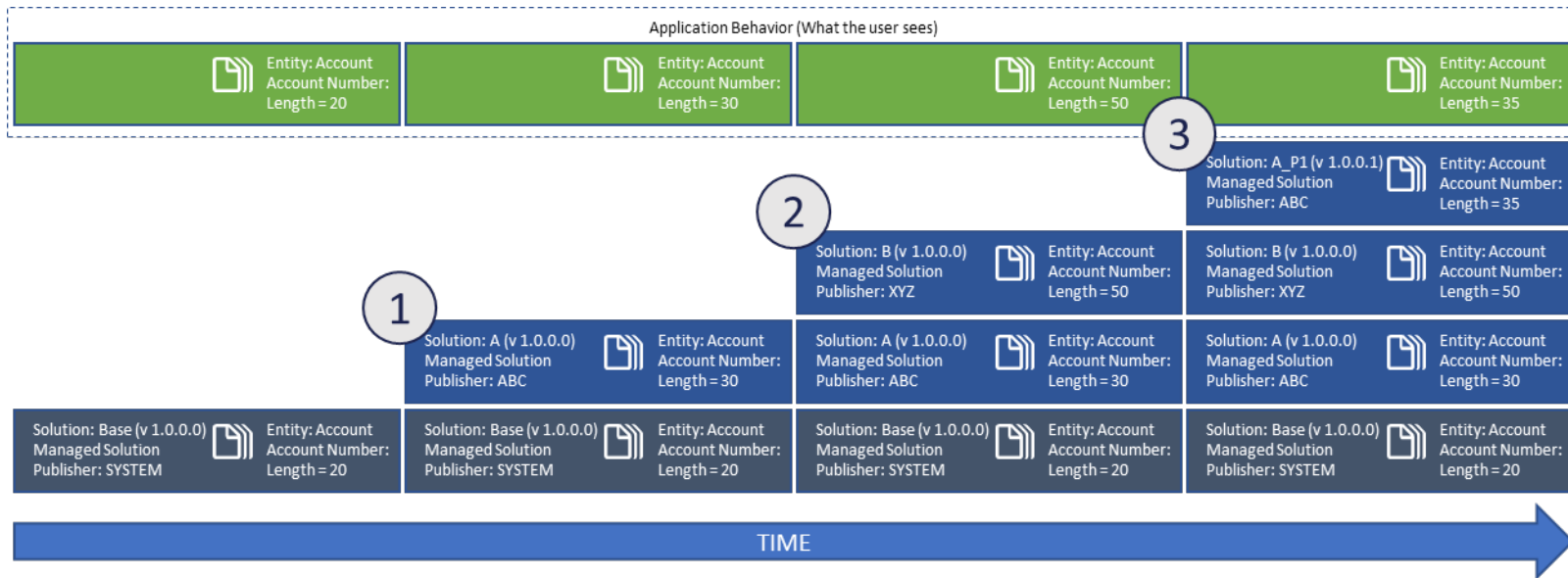


Figure 8 – Patching components by varying patch solution name

- 1) The Account Number field length is changed to 30. Solution A version 1.0.0.0 is imported. Application behavior reflects the change to Account Number field length.
- 2) Solution B amends the Account Number field length to 50. Solution B version 1.0.0.0 is imported and layers above Solution A based on the import time stamp. Application behavior reflects the change to Account Number field length.
- 3) The Account number field length is changed to 35. Solution A_P1 versions 1.0.0.1 is imported and layers above Solution B based on the import time stamp. Application behavior reflects the change to Account Number field length which may not be intended.

As can be seen above, creation of a new solution A_P1 to patch components imported through solution A results in the changes being imported to the top of the managed stack. This may not be intended behavior given that solution B had independently updated and masked the change imported by Solution A. The approach does not bind the patch solution to the original solution and the relationship is notional.

While this may be manageable in a simple implementation, it becomes more difficult to assess within a complex multi-stream implementation. If solution segmentation has not been fully utilized, this can lead to the introduction of additional and unnecessary

dependencies which can increase the complexity in assessing why changes are occurring and, in some instances, why they are not. The approach also masks changes introduced by Solution B (potentially inadvertently) which may have unintended behavioral consequences.

Formalizing patch management

The challenge with commonly observed patching approaches prior to release 8.0 stem for the inability to bind a patch solution to the parent solution. The consequences have to a degree explained why some enterprises continued to state a preference for unmanaged solutions. The main challenge with each of the three approaches (and any unmanaged solution approach), is the ability to apply patch changes predictably and in a manner that can be reversed. The analogy in a .NET world would be patching assemblies through an XCOPY process. The only approach to revert the change would be a further compensating XCOPY of an updated assembly.

Version 8.0 introduced the formal concept of patch solutions. These maintain a solution dependency to their parent solution and when imported, are added to the managed stack directly above their parent solution in version order. The analogy in a .NET world would be patching assemblies through an MSI install process. Changes can be predictably reverted by uninstalling the patch through the MSI uninstall process.

Patch solutions are documented in detail within the [Create patches to simplify solution updates](#) and [Use segmented solutions and patches to simplify solution updates](#) sections of the customer engagement developer documentation.

Figure 9 below illustrates how patch solutions interact with the system in a consistent and predictable manner.

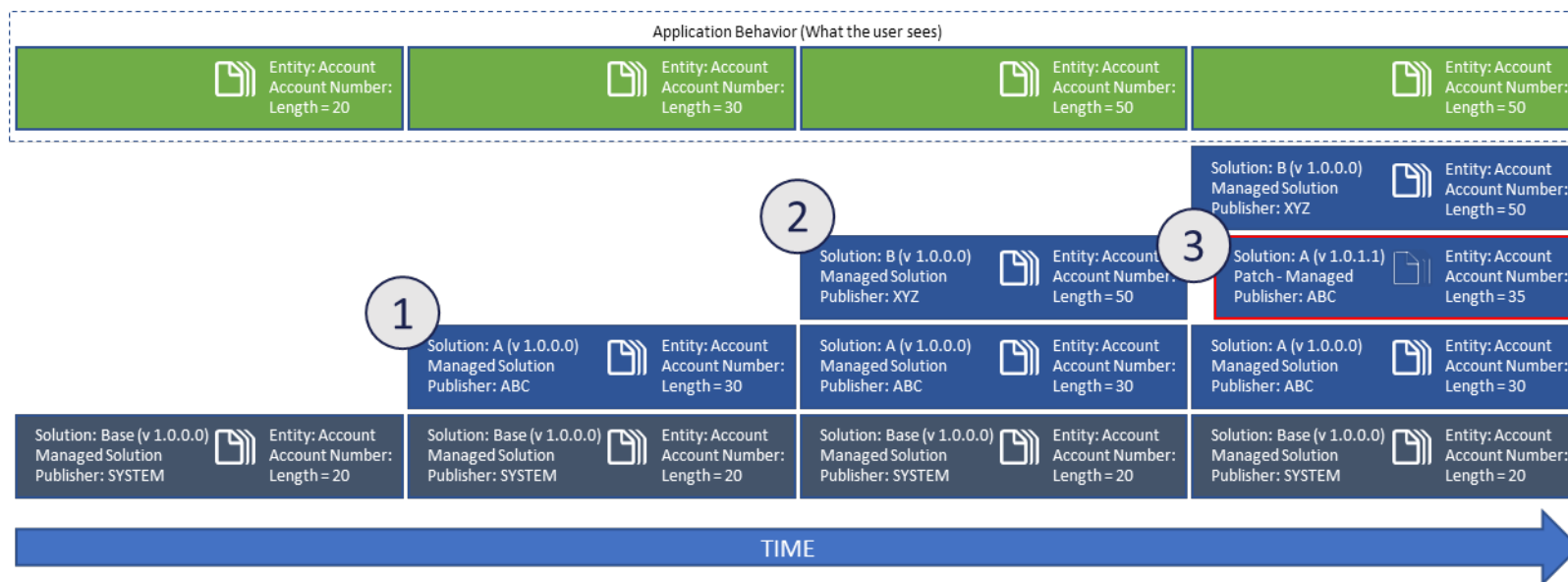


Figure 9 - Patching components correctly in V8.0+

- 1) The Account Number field length is changed to 30. Solution A version 1.0.0.0 is imported. Application behavior reflects the change to Account Number field length.
- 2) Solution B amends the Account Number field length to 50. Solution B version 1.0.0.0 is imported and layers above Solution A based on the import time stamp. Application behavior reflects the change to Account Number field length.
- 3) Solution A is cloned to a patch solution and the solution version number is incremented and the parent solution is locked. The Account number field length is changed to 35. Solution A_Patch_abc12345 version 1.0.1.1 is imported and creates a new layer directly above the previous layer for the parent, Solution A. Application behavior does not reflect the change to Account Number field length as Solution B is layered above and masks the change.

When combined with solution segmentation, adoption of the formalized patching process will provide predictable layering and application behavior. Additionally, the increased granularity in terms of what has changed enables improved version control to support modern agile implementation approaches.

Using patch solutions can also reduce the number and complexity of layers introduced into the target system. When the base solution is cloned to enhance the solution for the next release, all patch solutions are rolled up into the base solution and it becomes the next version.

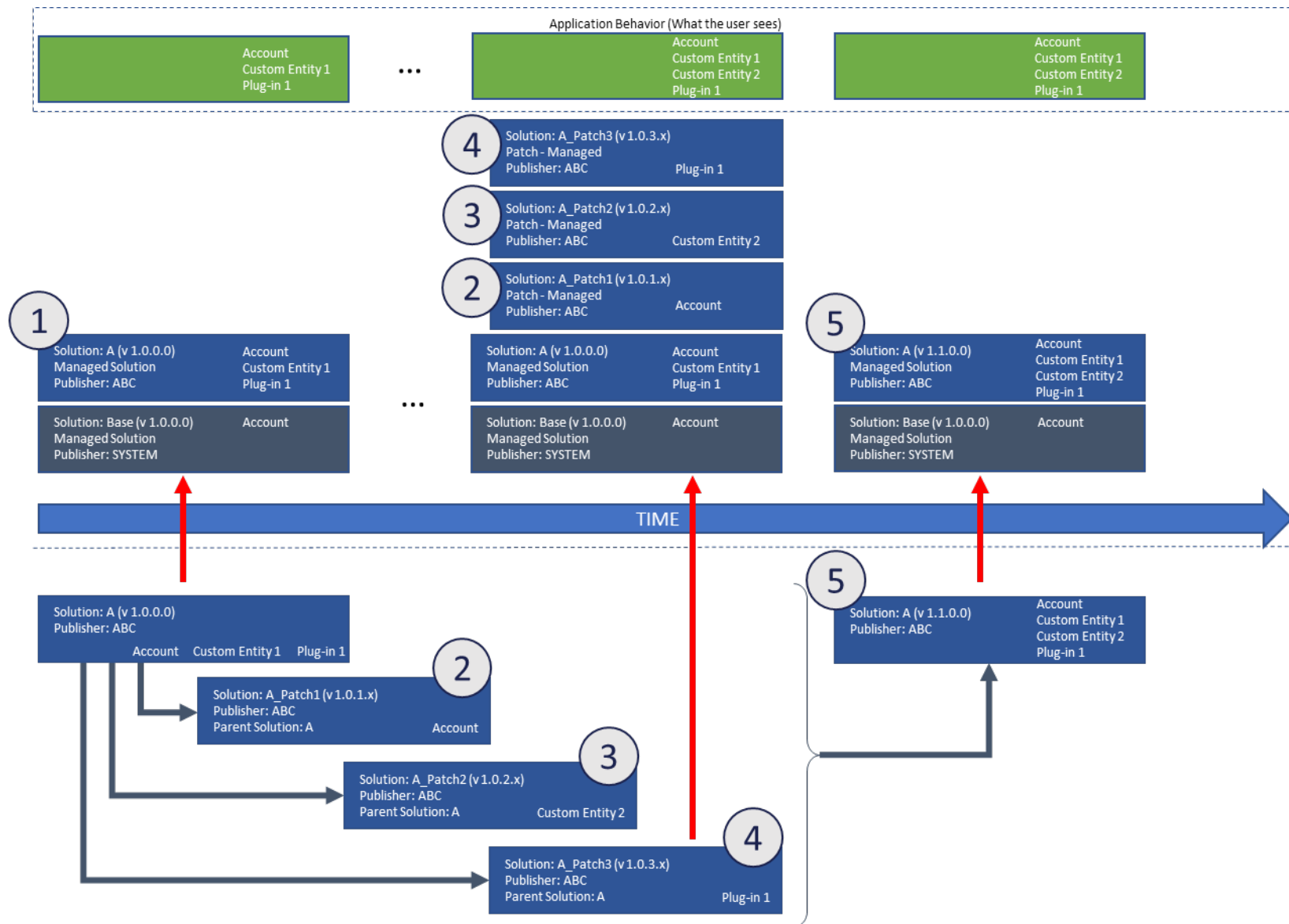


Figure 10 - Cloning base solution / rolling-up patches

- 1) Managed Solution A version 1.0.0.0 is imported into the instance and includes component customizations for the Account entity, a new entity (Custom Entity 1) and a plug-in
- 2) Solution A is cloned to a patch solution, the solution version number is incremented, and the parent unmanaged solution is locked. The Account entity is added to the patch and further amended. Solution A patch version 1.0.1.0 is imported and creates a new layer directly above the previous layer for the parent, Solution A. Application behavior reflects the changes to the Account entity as there are no layers above it.
- 3) Solution A is cloned to a patch solution and the solution version number is incremented. A new entity (Custom Entity 2) is added. Solution A patch version 1.0.2.0 is imported and creates a new layer directly above Solution A patch version 1.0.1.0. Application behavior reflects the addition of Custom Entity 2.
- 4) Solution A is cloned to a patch solution and the solution version number is incremented. An existing plug-in (Plug-in 1) is added to the patch and existing logic amended. Solution A patch version 1.0.3.0 is imported and creates a new layer directly above Solution A patch version 1.0.2.0. Application behavior reflects the amendment to Plug-in 1.
- 5) Solution A is [cloned as a solution](#) which creates a new unmanaged copy of the solution with version 1.1.0.0. The copy contains the components from Solution A plus all its patches. The new managed solution version is then imported ("Stage for Upgrade is enforced") and committed using the "Apply Solution Upgrade" button or via the [DeleteAndPromote](#) web request which upgrades the solution, replacing managed Solution A version 1.0.0.0 and its patches with managed Solution A version 1.1.0.0.



Important: Solution segmentation (only adding assets that are affected to the solution) helps to reduce the size and complexity of the solution imported which will result in reduced import times. Additionally, solution segmentation can reduce collisions caused by multiple team members working on assets of a component such as an entity.

Upgrade Solution versus Stage for Upgrade

When applying component customizations to an environment, there are two behaviors:

Upgrade Solution

Upgrading a solution is an atomic action. The managed solution is imported with a higher version number and layers directly above the previous version of the solution on a per component basis. This approach is used for straightforward component updates that do not need to delete components or perform any data migration.

Full solutions and patch solutions can be imported in this way and immediately upgrade the solution on import.

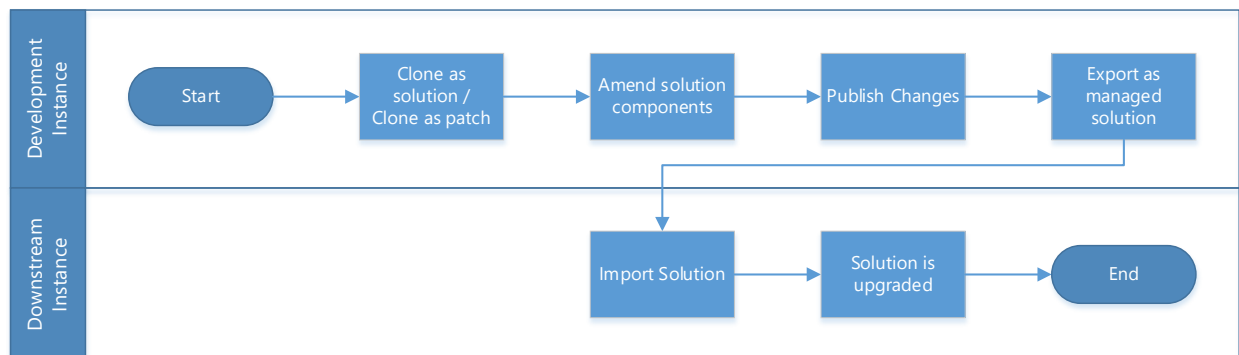


Figure 11 - Upgrade solution process flow

Stage for Upgrade

When importing a new major or minor version of a solution, “Stage for upgrade” imports the solution as a holding solution. This separates the import step from the apply step of the import, enabling two important capabilities and the ability to automate these capabilities:

- The ability to delete components
- The ability to conduct data migration on the upgrade before applying the solution

Stage for upgrade is mandatory for managed cloned solutions that are imported on top of patch solutions to protect against configuration and data loss when replacing the patches and their parent solution with the new managed solution version.

Deleting customizations and components

Configuration within Dynamics 365 CE is additive which presents a challenge when it is necessary to perform a deletion.

Components are only preserved for as long as there is at least one reference to them. The managed solution that introduced the component to the system owns the component. Additionally, there may be other managed solutions layered above this component that take a dependency on it. Assuming any higher layer component dependencies have been removed (which is a good reason to use solution segmentation to avoid unnecessary dependency creation), the owning solution still needs to be addressed.

There are two considerations:

- 1) It is not possible to delete the entire owning solution as other components within the solution continue to be required and it would result in data loss
- 2) It may be necessary to perform a data migration from the component to be removed in order to avoid data loss (i.e. changing the data type requires addition of a new field, data migration from the original field, followed by removal of the original field)

Managed components are only deleted when they are no longer referenced. The holding solution technique enables the final reference to the component to be removed without removing all other components that only have a singular reference contained within the solution. The approach uses a holding solution that contains all the components of the solution, minus the components that need to be deleted. The holding solution is imported which increments the reference count for all components it contains. The solution in which the components to be deleted reside is deleted which decrements the reference count for all components it contains. This results in a reference count of zero for the components to be deleted and they are deleted from the system. Optionally, the holding solution is renamed to maintain naming consistency with the solution it replaced.

Two behaviors were introduced to simplify the process through the API and user interface:

[CloneAsSolution](#) - Creates a duplicate copy of an unmanaged solution that contains the original solution plus all its patches.

The same behavior is also available directly through the user interface via the "Clone Solution" button on the solutions grid.

[DeleteAndPromote](#) (Apply Solution Upgrade)- Replaces managed solution (A) plus all its patches with managed solution (B) that is the clone of (A) and all its patches. For DeleteAndPromote to be effective in this manner, requires the solution to have been imported with "Stage for Upgrade".

The same behavior is also available directly through the user interface via the "Apply Solution Upgrade" button during solution import or from the solution grid

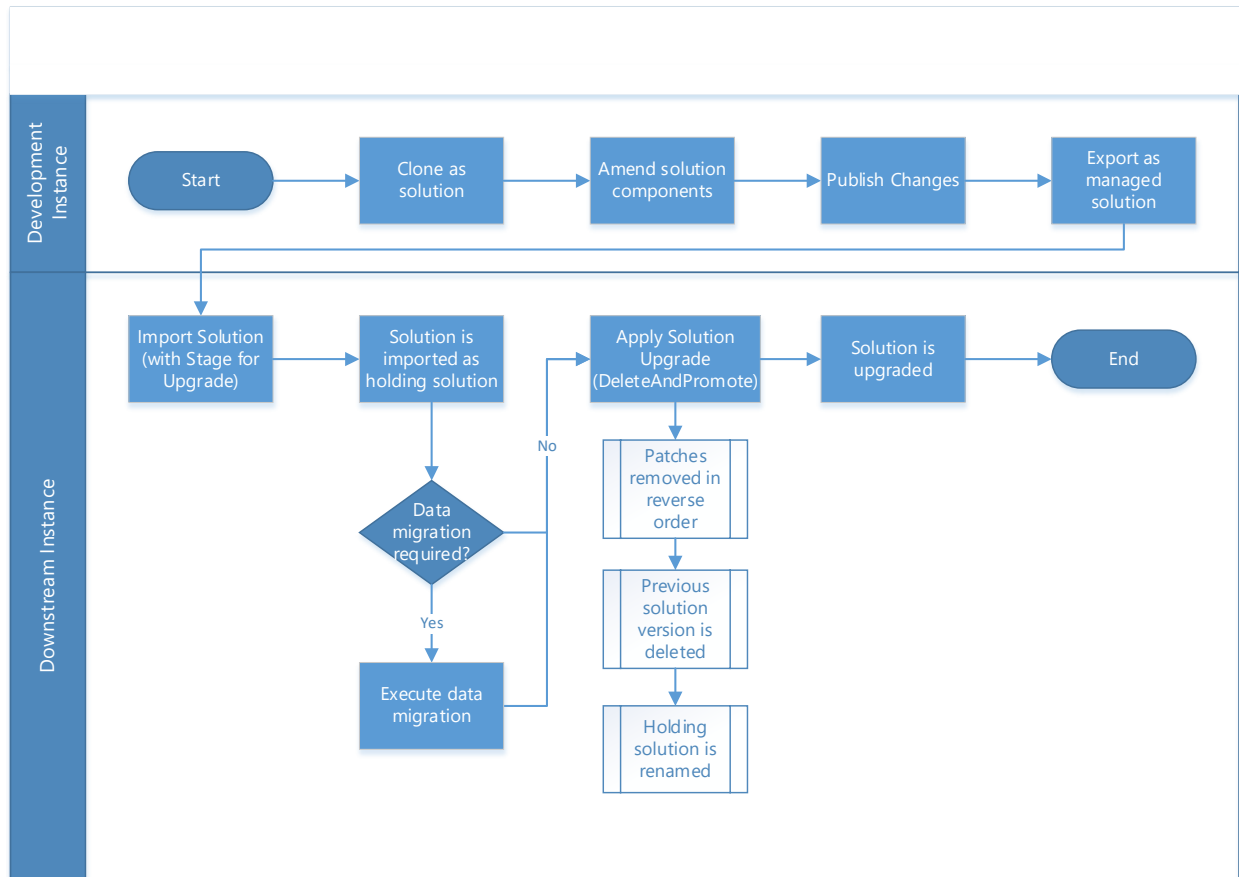


Figure 12 - Stage for Upgrade process flow

Why is this important?

Separating the logic contained within the [DeleteAndPromote](#) action from the logic contained within the [ImportSolution](#) action, enables the holding solution technique and component deletion. It also provides the opportunity for a data migration process to be executed as part of the process to ensure implementation teams can avoid data loss. Additionally, the [Package Deployer](#) tool supports the ability to execute custom code during or after the package is deployed, enabling automation of the process. See [Developer Tooling](#) below for more details.

Plugin-types and versioning

Prior to version 9.0, developers had the ability to fundamentally amend plug-ins and custom workflow activities. This caused import and deletion failures due to missing dependencies. Version 9.0 introduces constraints on when new plug-in types can be introduced (based on assembly version number) and how they will be blocked on import. New types may only be introduced when the major or minor assembly version is changed.

Through this change, unnecessary and missing dependencies will be avoided which will alleviate import and deletion failures caused by plug-ins with the net result of a more predictable and repeatable deployment.

Composing applications

Before addressing lifecycle and deployment approaches, it's important to consider what type of application is being delivered. Applications typically fall into the following categories:

- **Complete Applications** – That provide an encapsulated and often vertical set of capabilities. They may depend on other Dynamics 365 CE system functionality or in the case of applications built directly on the Common Data Service (CDS) for apps, they may be completely independent. Field Service and Project Service are examples of first party complete applications
- **Applications that Dynamics 365 CE is part of** – for example, a broader citizen services platform that also combines a suite of IoT embedded sensors and wirelessly connected devices
- **Feature additions to Dynamics 365 CE** – That augment existing capabilities or provide a breadth set of capabilities. Portals and Microsoft Dynamics Apps such as LinkedIn Sales Navigator illustrate this application type
- **Task based apps** – The use of PowerApps is becoming prevalent to deliver experiences either for Dynamics 365 or for CDS for Apps directly. In addition to democratizing development, PowerApps can be considered in a similar manner to the feature additions category in that they augment existing capabilities and fill implementation specific whitespace through tailored user experiences, often used to focus on a sub-set of business tasks.

Depending on the type of application being built, the application could involve major components that are external to Dynamics 365 and the solutions framework. It is likely that most implementations will involve integration with other systems and therefore can be categorized in this manner.

It's important to consider the lifecycle of each component used to compose the application. The following sections focus on those components delivered through the solutions framework on the Dynamics 365 CE platform.

Defining solution boundaries

There are a variety of approaches to defining solution boundaries and there is no right or wrong answer. What is important is to ensure the needs of the organization are addressed in a manner that works today and provides flexibility for future updates and enhancements. How this is achieved will likely vary depending upon the type of organization delivering the solution:

- Independent Software Vendor (ISV)
- Implementation Partner
- End Customer organization

There are multiple factors to consider when defining solution boundaries. These are based on deployment models (single global instance versus multiple regional instances), time independence for varying a sub-set of application behavior (e.g. sales versus service deployment), and the level of isolation between team members during development.

Given that solution packages are the unit of transport for application behavior between environments, it can help to consider the solution as the atomic boundary that discretely encapsulates a unit of application behavior.

A practical approach to assessment is to consider the needs in production and to work backwards through environments. Factors that often influence solution boundaries include:

Table 4 - Factors influencing solution boundary definition

Production Deployment Model	Single instance versus multiple instances
<p>Will the application be deployed to multiple discrete customers? Potential Outcome: A single "shrink wrapped" solution delivered through AppSource</p> <p>Will the application be deployed to multiple regional instances?</p> <ul style="list-style-type: none"> • Implies there is a need for consistency across regions, but other factors exclude a single global deployment (for example, regulatory). • May require a degree of flexibility to regionally vary application behavior (in terms of functionality or regional systems to integrate with) <p>Potential Outcome: A single "core" solution with flexibility for regional dependent solutions that extend/augment application behavior.</p> <p>Note: whether the core solution is one or more "core" solutions will be dependent on other factors including time independent variability below</p> <p>Will the application be deployed to a single global instance? Potential Outcome: A single solution (dependent upon time independent variability below)</p>	
Time Independence	Varying a sub-set of application behavior, time independently
<p>Sometimes, units of application behavior will have discrete iteration timelines. This could represent an entire vertical, a capability area or specific feature. The assessment should consider the benefit of isolating sub-sets of application behavior into separate solutions to enable independent delivery versus the additional complexity of managing solution boundaries</p> <p>Will the application work if behavior is split? If not, adopt a single solution approach</p> <p>Is there a need or benefit (now or in the future) to releasing a sub-set of application behavior on a different cadence?</p>	

Is there clear demarcation between sub-sets of application behavior for example by usage scenario or user role?

Are there enough Dynamics 365 for Customer Engagement instances available to support isolated implementation of sub-sets of application behavior on independent timelines?

Configuration and version control

Enabling agile teams to deliver while minimizing collisions

Implementation teams typically have different solution needs in development when compared to the ultimate solution packages deployed into test and production environments. Team members will likely require a degree of isolation from each other to avoid impacting throughput. Introducing isolation however, may increase the incidence of collisions when merging changes.

The published documentation covers options in full within this article: [Organize your team to develop solutions](#)

The important point to note is that solution boundary definition for downstream environments should be based on the production need rather than the being defined by the development need. This includes the production requirements for governance, removal, patch and upgrade of application behavior

How many instances are required?

The number of solutions used to deliver an application will naturally have an impact on the number of Dynamics 365 for Customer Engagement instances necessary to build and maintain them. Equally, it is important not to neglect that once an application is live, there is likely to be a need to support implementation of the next significant application release version in parallel to maintaining the current live version. This may lead to a requirement for additional Dynamics 365 for Customer Engagement instances.

Consider the scenario where an application comprises of a single solution. Version 1 implementation takes place and is released to production.

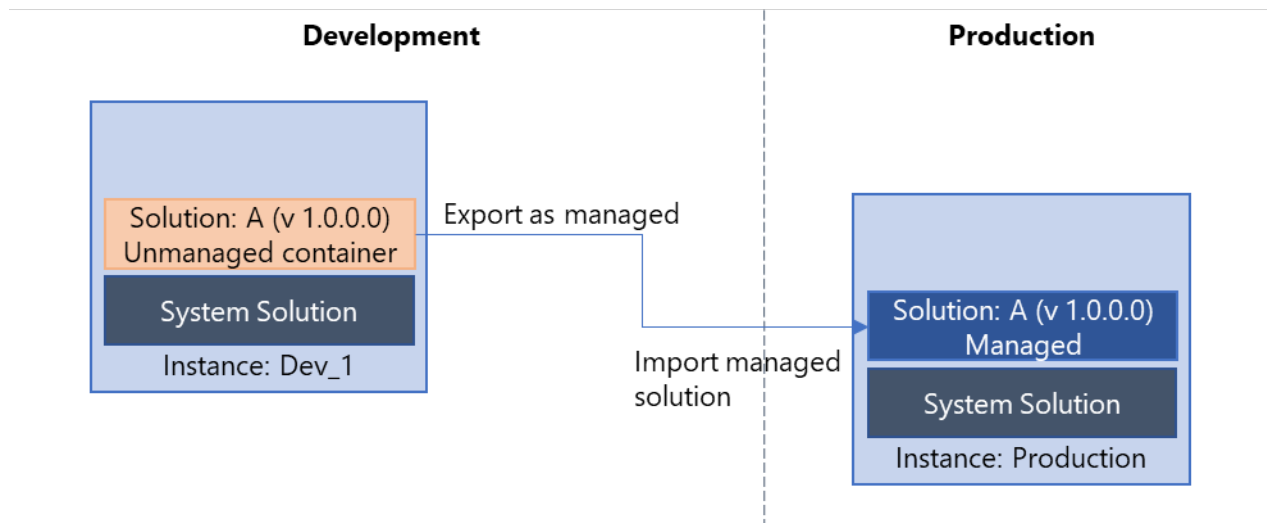


Figure 13 - Supporting a single version/single solution application

The development instance is preserved to support any necessary patching of the live application. In parallel, another development instance is required to enable Version 2 implementation work to proceed. Patches to Version 1 are incorporated with enhancements for Version 2 by propagating as unmanaged changes between development instances.

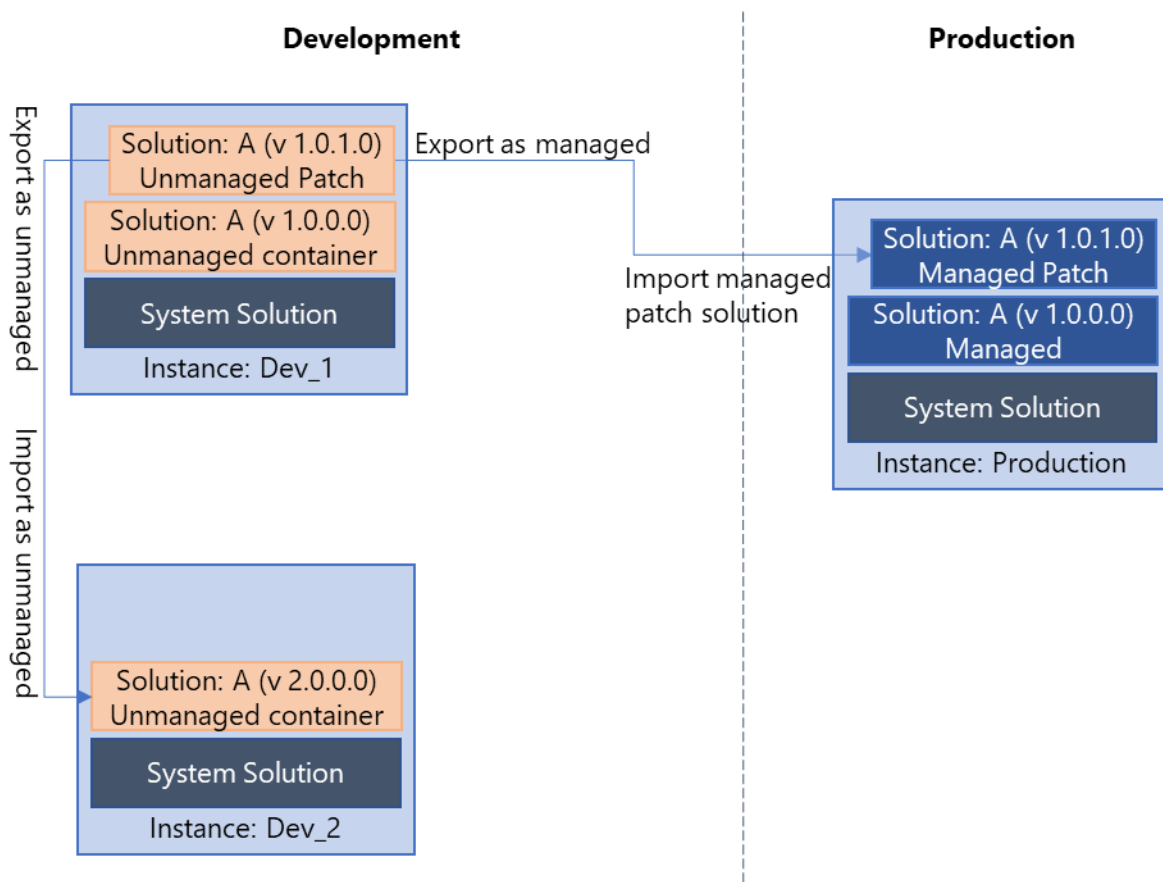


Figure 14 - Supporting current and next versions of a single solution application

When Version 2 implementation is complete and releases to production, the development instances are rotated. Dev_2 is preserved to support any necessary patching of the live application and Dev_1 is recycled to commence Version 3 implementation work in parallel.

Expanding this scenario to consider an application comprising of two solutions, one dependent upon the other, it is possible to extrapolate that four development instances may be required. Consider the example of a core solution, with a regional variant solution dependent upon the core solution. The single solution example above has demonstrated that two development instances will be required to maintain Version 1 of the core solution in live use while Version 2 undergoes implementation in parallel. Given the regionally tailored solution has a dependency on the core solution, a third instance is required to enable the dependency on the core solution to be satisfied by importing the managed version of the core solution.

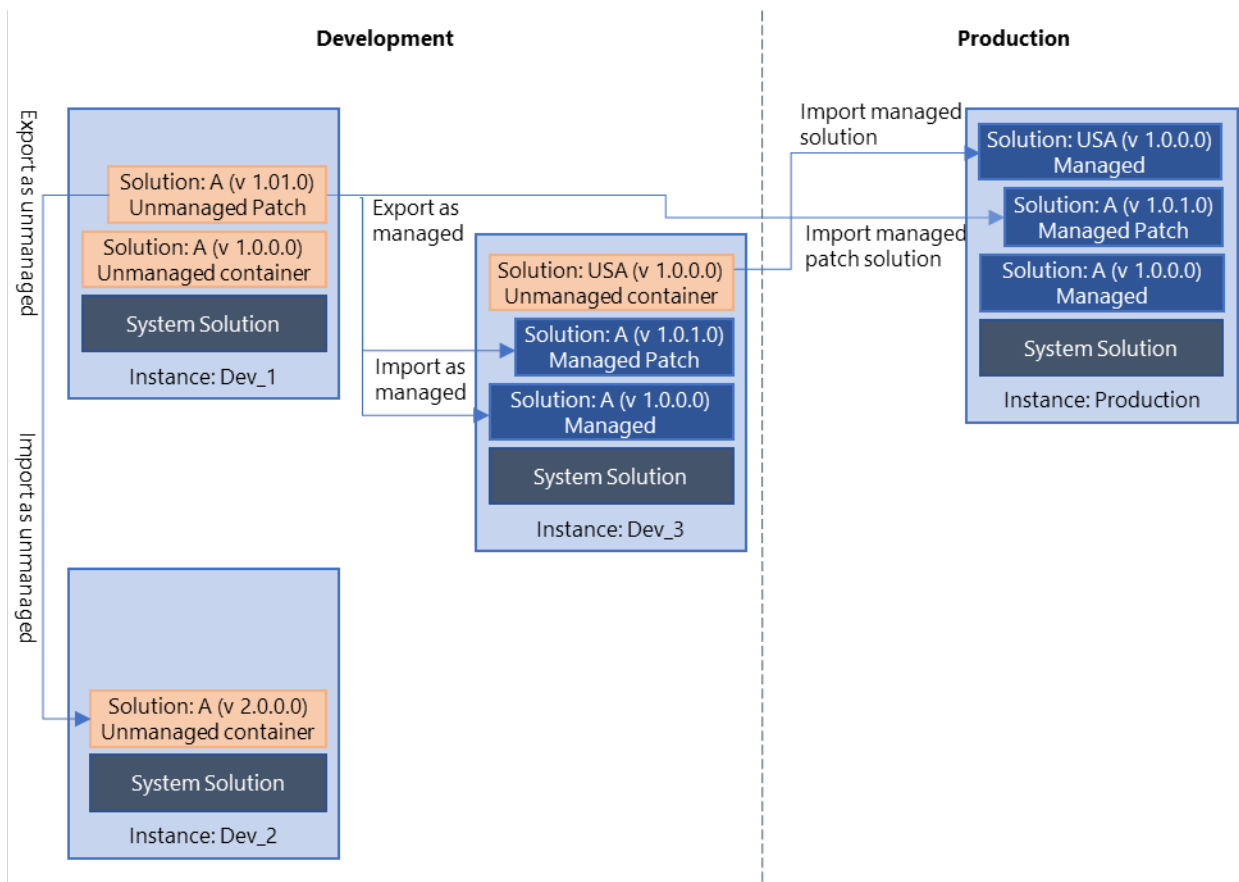


Figure 15 – Extending to support a multi-solution application

Finally, there is potential for regional variations to be implemented on an independent timeline to that of the core solution. This introduces a need for a fourth instance to support Version 2 of the regional solution to be implemented while Version 1 is maintained in live use.

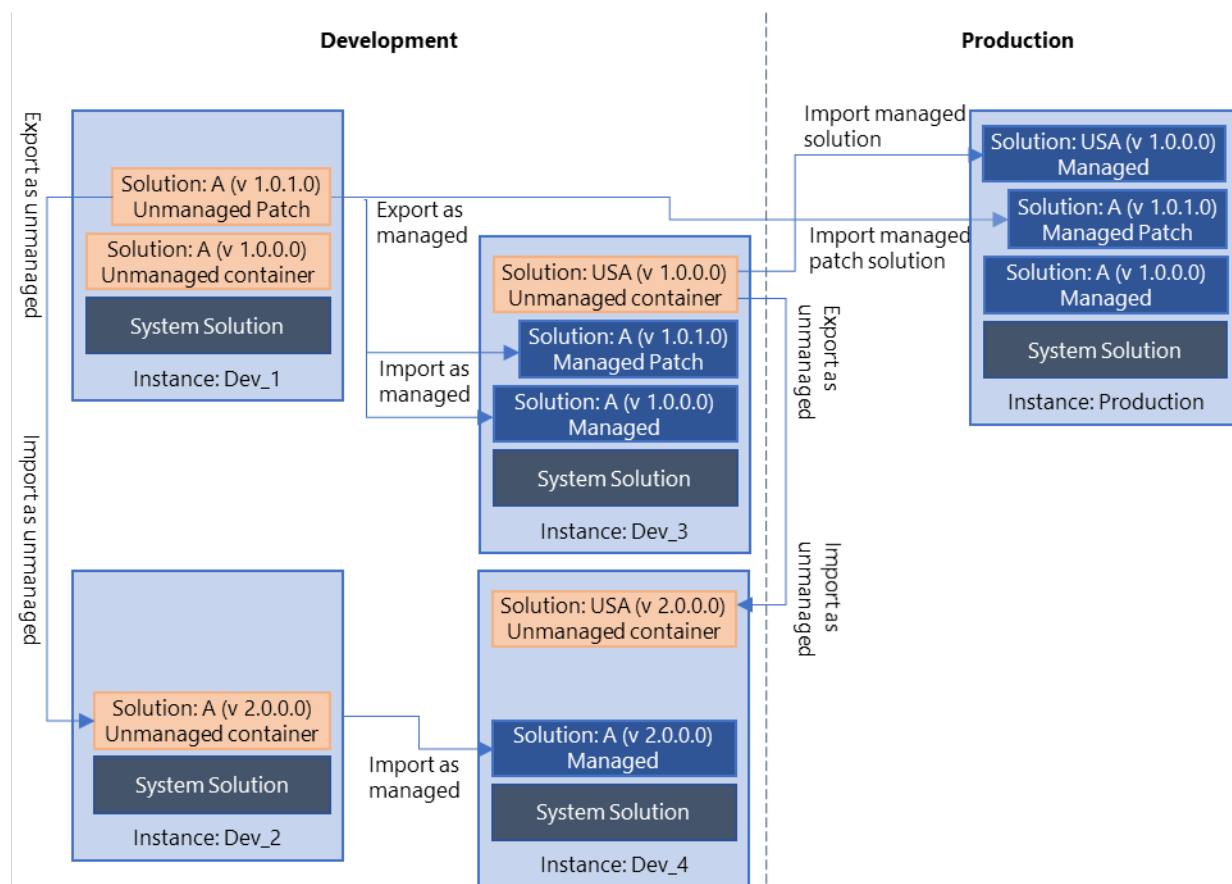


Figure 16 - Supporting current and next versions of a multi-solution application



Important: The number of Dynamics 365 for Customer Engagement instances required to support development of an application is a function of the number of discrete solutions being deployed to production.

As a rule of thumb, the typical number of Dynamics 365 for Customer Engagement instances required for development of solutions is twice the number of discrete solutions being deployed to production. This illustrates why it is important to define solution boundaries based on functional production need to avoid introducing artificial instance requirement.

The calculation is based on several assumptions and does not include instances used for test or production purposes. Multiple solutions can co-exist and be developed within a single development instance when the assets they contain are entirely isolated from each other. This is more likely to occur in xRM/CDS for Apps based implementations that do not take any dependency on the Dynamics 365 for Customer Engagement core applications.

Assumptions:

- Instance backup/restore is not used to “flip-flop” between two versions of the same solution
- Current production solution is maintained/patched in parallel to the next major/minor version of the solution being developed
- If multiple solutions are being developed:
 - Either
 - A solution is dependent upon another solution, requiring deployment in a managed state to the development instance
 - Or
 - Solutions have independent iteration timelines and require isolation, but they affect common components (for example ribbons)

Solution Lifecycles

Repeatable and predictable deployment

Within enterprise implementations it is normal to have governance and release management processes to control what, when and how solutions are deployed – and if necessary, rolled back. It is beneficial to test the deployment processes for production as early and as often as is possible. Consider the following environments:

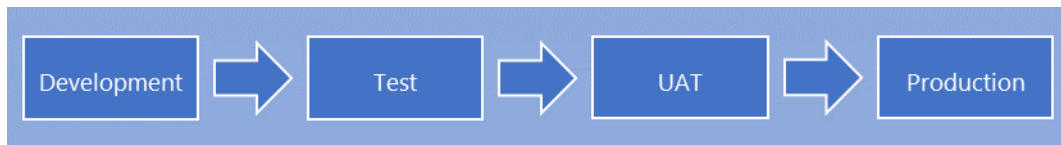


Figure 17 - Solution propagation through environments

To gain confidence that the deployment approach for production is repeatable and predictable, it is typical to take the same deployment approach for environments upstream of production. The earlier within the cycle the final deployment approach can be tested, the more opportunity there is to resolve issues. Deployments can be fully automated to remove potential human error and further ensure repeatability and predictability. See [Tooling for automation](#) later in this document.

The proceeding sections cover the considerations to progress solutions through environments and achieve a predictable and repeatable outcome.

Development approach

At a high level the development processes for solutions is relatively straight forward.

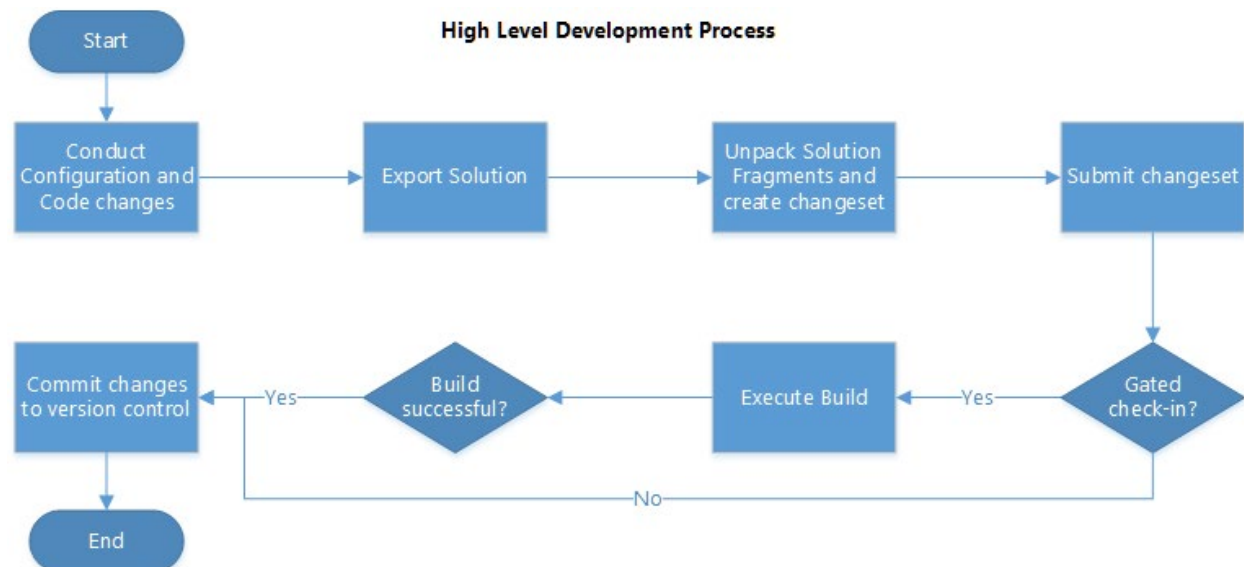


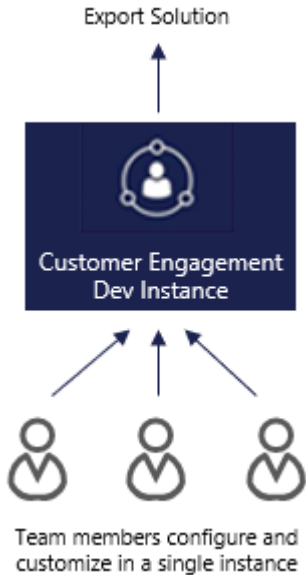
Figure 18 - High level basic development process

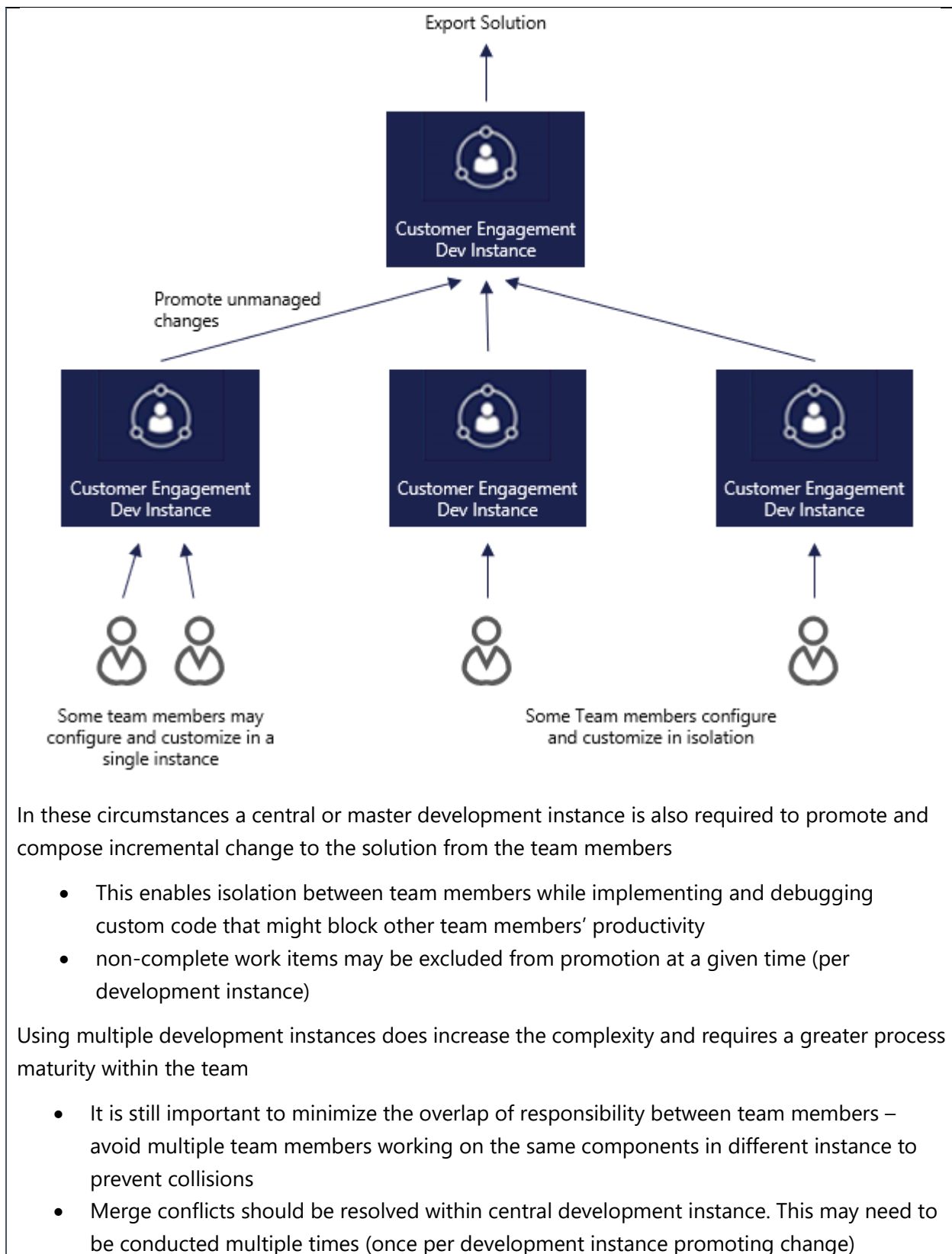
The primary challenge occurs when there is a need to merge configuration changes which requires a Dynamics instance. There are several working patterns commonly observed during development.

Consider the following patterns in the context of developing a single solution. Multiple instances may be required to support development of dependent solutions for the reasons covered in Table 4

- Factors influencing solution boundary definition

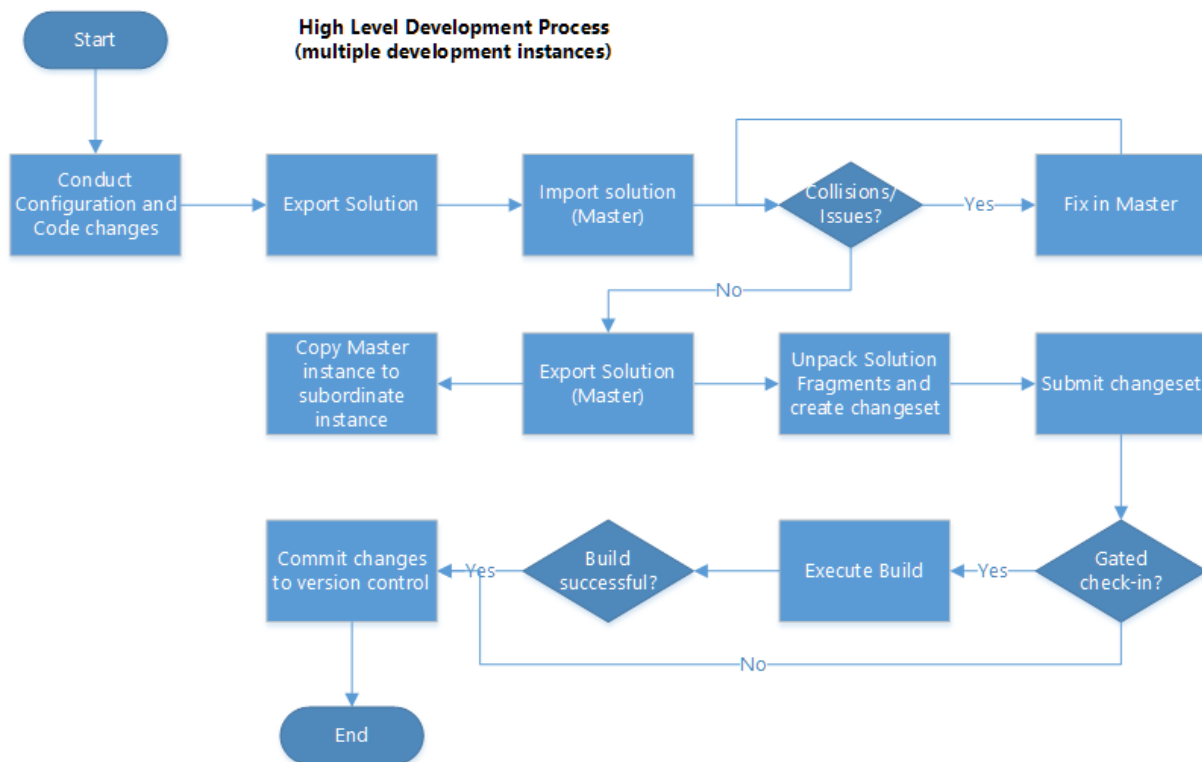
Table 5 - Instance topologies for development approaches

Single development instance	
All team members configure and customize functionality within the same environment	
 <p>Export Solution</p> <p>Customer Engagement Dev Instance</p> <p>Team members configure and customize in a single instance</p>	<p>For less complex deployments or where the team is relatively small, a single development instance is the most practical approach. It avoids merge complexity and the effort to automate version control, build and deployment is relatively low.</p> <p>There are challenges to be aware of and manage:</p> <ul style="list-style-type: none"> • Minimize the overlap of responsibility between team members – avoid multiple team members working on the same components • Ensure Work Items are suitably granular and regularly committed – When the solution is exported, the components included within the solution will contain all changes made to them at that point (whether they are complete or not). Using solution segmentation can help to limit the impact but good team governance processes are required.
Multiple development instances – single workstream	
All team members work towards composing a single solution for release. They require a degree of isolation from one another to avoid contention and bottlenecks during the development phase	
<p>Typically for more complex deployments, a single development instance does not provide the agility required for an efficient development cycle. Remember that production deployment models should drive the number of solutions required, but even when all the team are working towards a single solution deployment, multiple development instances may be required (up to 1 instance per team member).</p>	



- Once changes are promoted to the central development instance, the subordinate instance should be "reset" to be aligned with the master. This can be achieved with an instance copy for online or a DB Backup/Restore for on-premises
- Ensure Work Items are suitably granular and regularly committed. This minimizes the differences between sub-ordinate development instances and will help to reduce merge conflicts and collisions

The basic development process requires extension as highlighted below

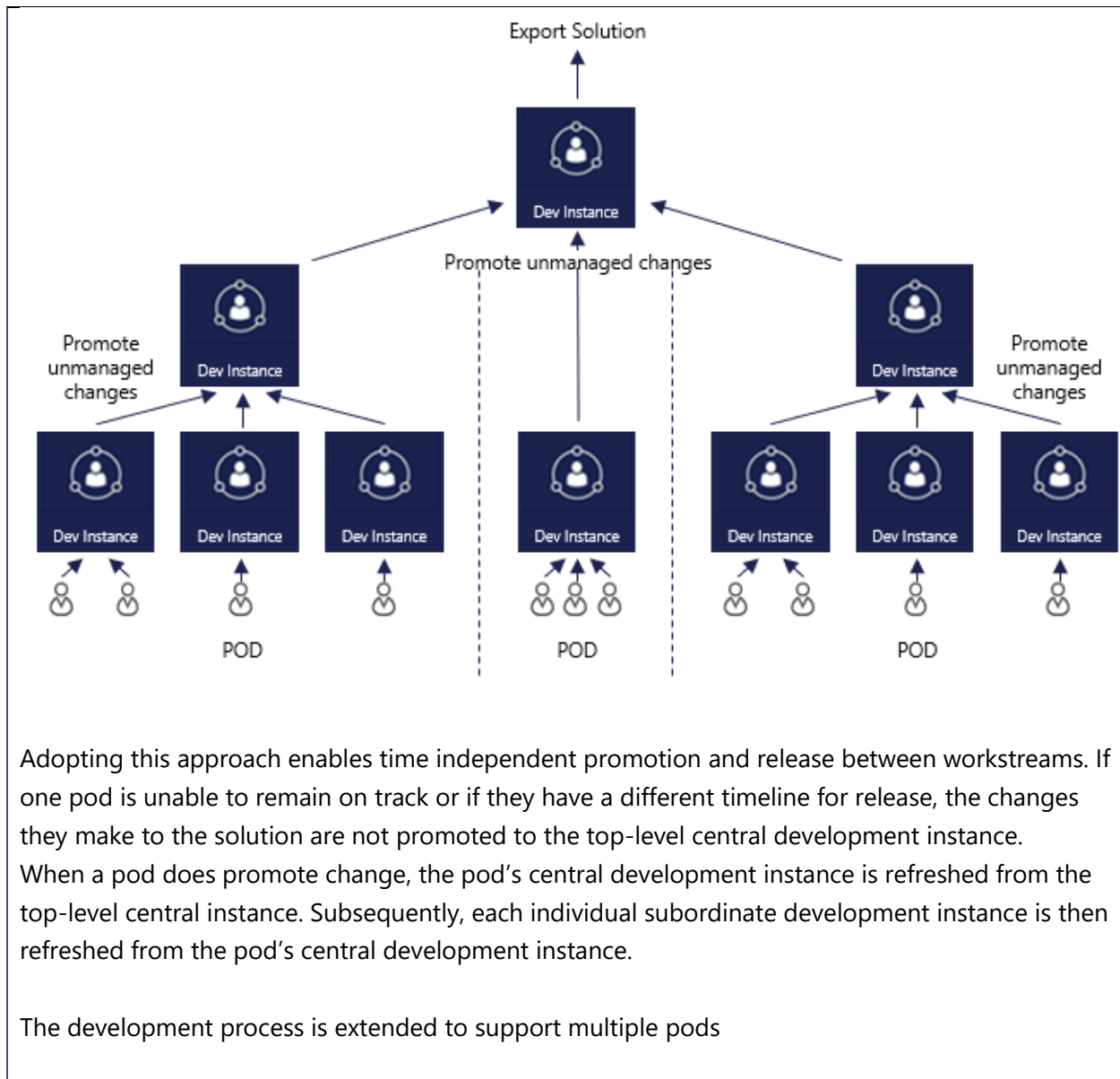


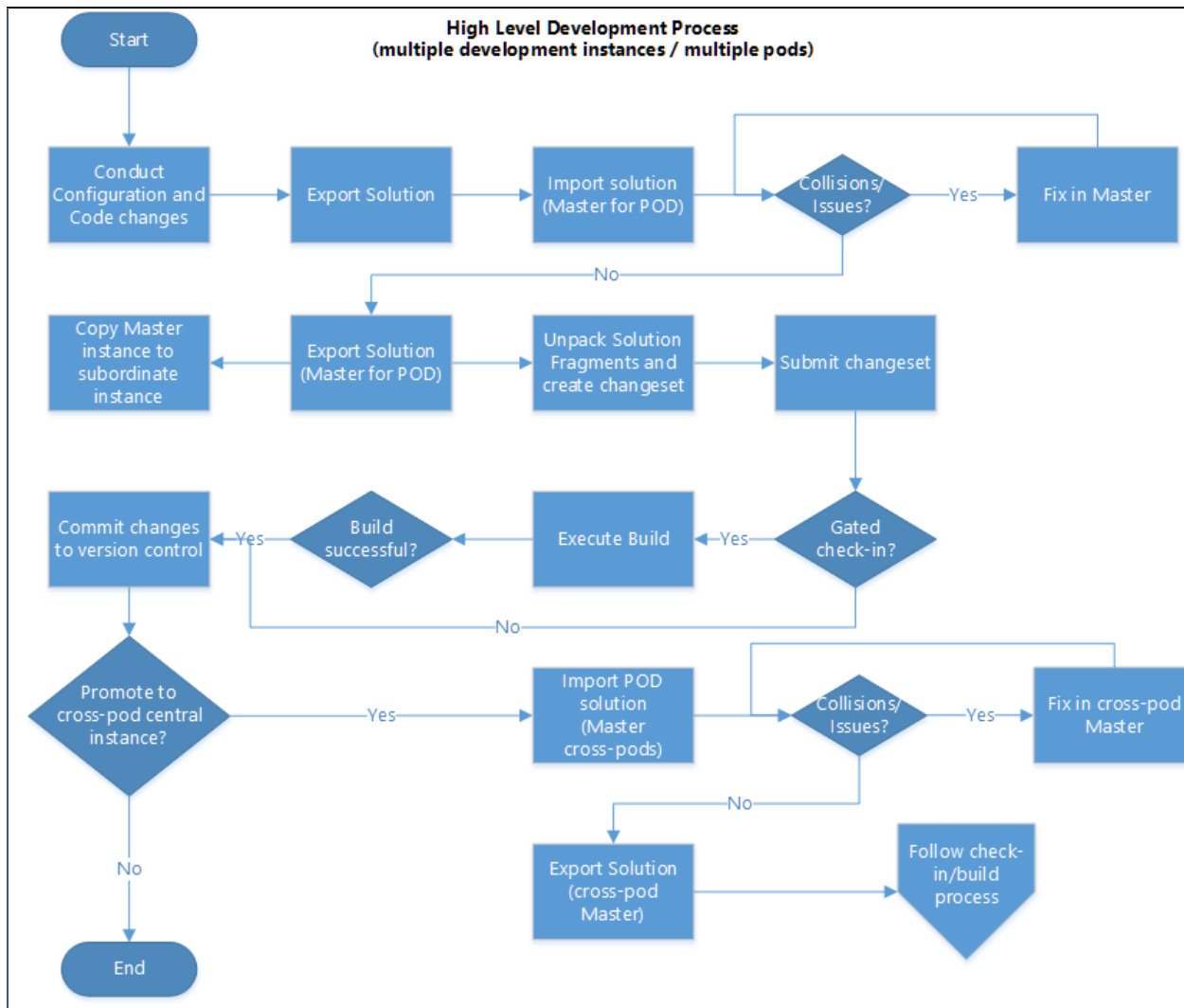
Multiple development instances – multiple workstreams

Team members are organized into pods (for example sales pod and service pod). They may need to work on the same solution (for example a common core solution) but require isolation from other pods for stability or for time independent release.

This approach is typically reserved for the most complex deployments and for ISV solution builders. A higher level of process maturity is required than the previous approaches.

The approach is based on extending the single workstream approach, replicating the structure and adding an additional master instance to compose and merge changes across pods. In this approach, typically each pod will have multiple instances, but they could equally have a single instance.





The above approaches illustrate the two ends of the spectrum from simplicity through flexibility. It's important to note again that these represent the process for a single solution. When a deployment comprises of multiple dependent solutions, the environments illustrated above would multiply as described in the [How many instances are required?](#) Section above.

Where flexibility is required, it may appear that complexity increases significantly. This is not necessarily the case, particularly where good automation techniques are adopted to avoid manual overhead. This is also true when debating a potential DevOps concern when assessing fixing in production versus the time to propagate a change through environments.

Tooling for automation

The platform exposes web services supporting the solutions framework. These are directly available and enable organizations to create custom-made automated build and release processes. ISVs,

partners and community members also provide 3rd party tooling that leverage these capabilities. A suite of tools is also available as part of the SDK that wraps the web services that are exposed as part of the platform's solutions framework. These tools are available through NuGet

<https://docs.microsoft.com/en-us/dynamics365/customer-engagement/developer/download-tools-nuget>

The most common Microsoft tools to adopt to support automated build and release processes are highlighted below.



Important: The tools available from NuGet have been provided standalone such that they can be used with a variety of Integrated Development Environments (IDEs) and build automation systems.

Solution Packager

Documentation Link: <https://docs.microsoft.com/en-us/dynamics365/customer-engagement/developer/compress-extract-solution-file-solutionpackager>

The Solution Packager enables extraction and packing of a solution.zip file and is commonly used as part of the check-in and build processes.

Creating a granular change set and committing

Using the Solution Packager with a **/action: Extract** switch unpacks the solution into XML Fragments that are laid out in a granular folder and file structure. The tool enables both unmanaged and managed versions of the solution to co-exist within the same folder and file structure. This capability provides the ability to commit a change set that can later be used to build unmanaged and managed versions of the solution and thus support further use for development and for downstream environments.

Only those XML fragments that have changed will be checked in which provides better visibility in terms of who changed what and when.

Note: It is still important to use [solution segmentation](#) to minimize the assets that are later packed into the deployable solution.

The process flow below highlights the steps and options to automate a check-in

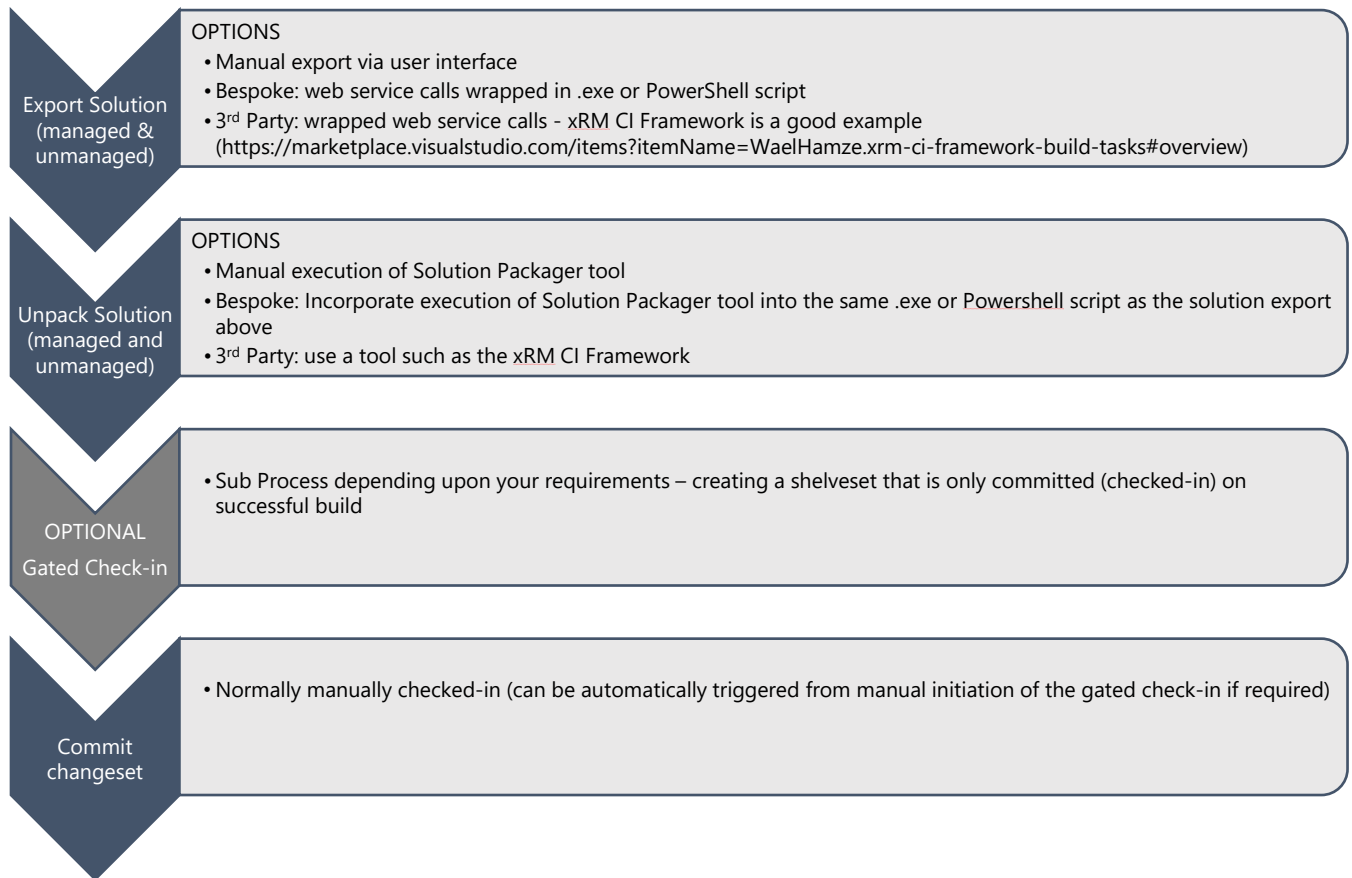


Figure 19 - Automating a check-in

Refreshing binaries and producing a drop/release

The Solution Packager is also used to pack the solution fragments back together into a solution that can be imported into an environment. The tool provides support to pack an unmanaged solution, a managed solution or both variants. Creating both solution types requires that you have unpacked both solution types in the same folder.

It is often the case that the solution will contain binary files representing plugins and custom workflow activities. These binaries may require building and refreshing prior to packing the solution. The Solution Packager provides the **/map** switch to support copying the refreshed assemblies into the folder structure prior to packing the solution.

Build automation systems are typically extensible and provide the ability to call external tools pre or post a specific build activity. In the build process, the Solution Packager is invoked to pack the solution post compilation and prior to generating a drop or release.

The process flow below highlights the steps and options to automate a build. This focuses on the aspects to produce a solution but could equally incorporate other assets required by the application.

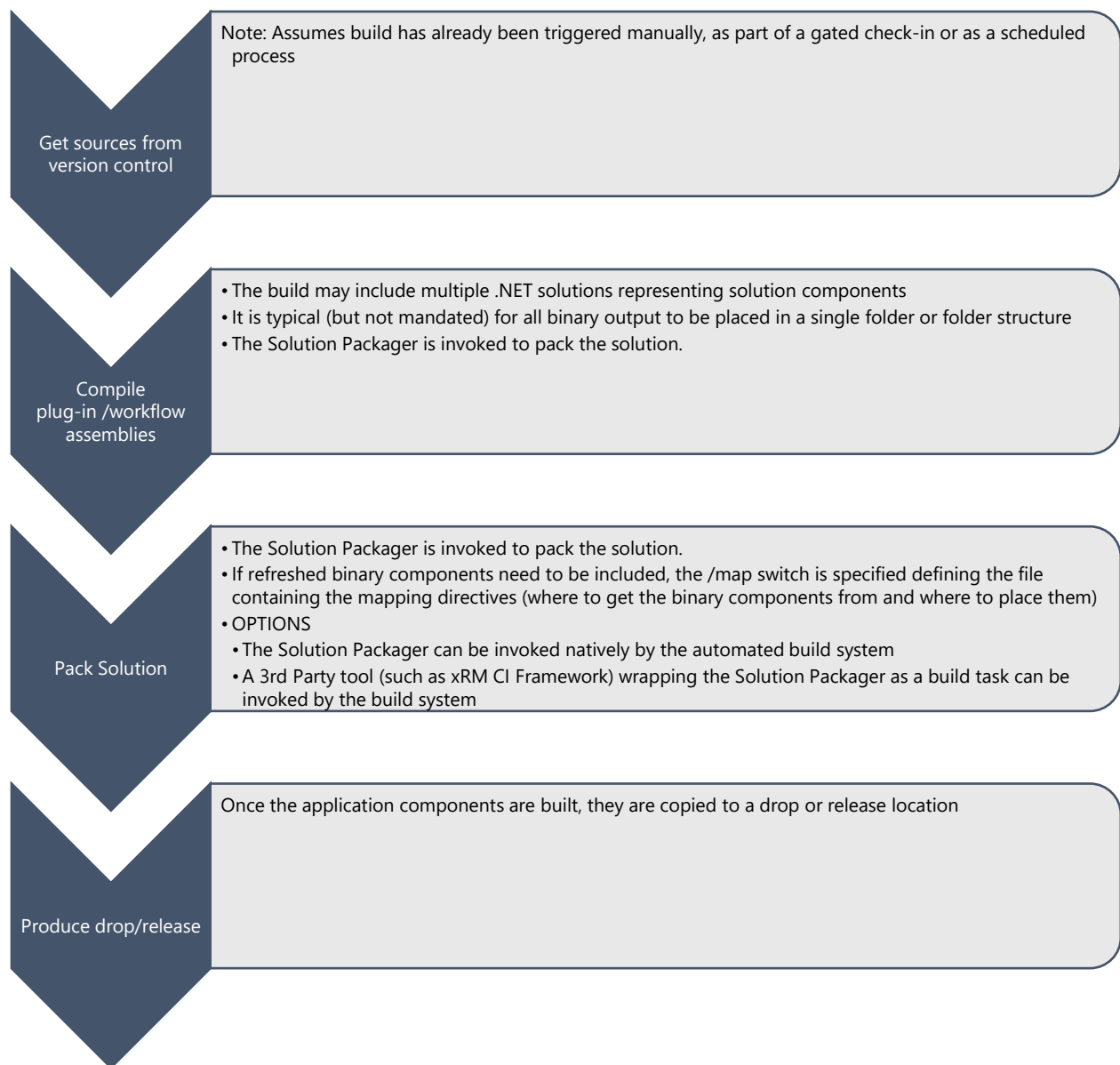


Figure 20 - Automating a build

The build may be extended to incorporate steps to compose a package for deployment through the [Package Deployer](#) tool covered later in this section.

Configuration Migration Tool

Documentation Link: <https://docs.microsoft.com/en-us/dynamics365/customer-engagement/admin/manage-configuration-data>

Most implementations require reference data or other configuration data to be present for the application to be provisioned in a repeatable and predictable manner. The Configuration Migration tool provides the capability to move configuration data between Dynamics 365 for Customer Engagement instances. Importantly, each step is decoupled, allowing the schema and data to be

version controlled, output as part of a build and later deployed as part of a package via the [Package Deployer](#).

Package Deployer

Documentation Links:

<https://docs.microsoft.com/en-us/dynamics365/customer-engagement/developer/create-packages-package-deployer>

<https://docs.microsoft.com/en-us/dynamics365/customer-engagement/admin/deploy-packages-using-package-deployer-windows-powershell>

Enterprise implementations are likely to compose applications from more than one solution and require reference data to be deployed in a seamless manner. Patching a live release is likely to require automation to ensure predictability and reliability and, in some cases, where components are being removed or amended, the ability to conduct a data migration or transformation as part of the process.

The Package Deployer assists with these processes and enables deployment of a package that can contain multiple solutions, reference data, and execute custom code during or after package deployment. The tool can be launched interactively, executed from the command line or through PowerShell for automated deployment.

A [Visual Studio project template](#) is provided to construct package definitions. The package definition holds the assets to be deployed (solutions, data files, custom code to execute during deployment) and defines the characteristics of the deployment (solution deployment order, when to trigger custom code).

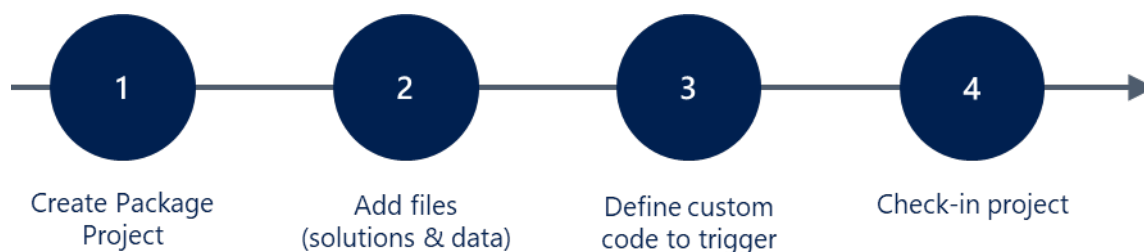


Figure 21 - Creating a package project

The project can be version controlled and can form part of the build process – taking the solutions output from solution packaging as input for package deployment. This is analogous to building an MSI that can later be installed. The diagram below illustrates the logical extension of the build process.

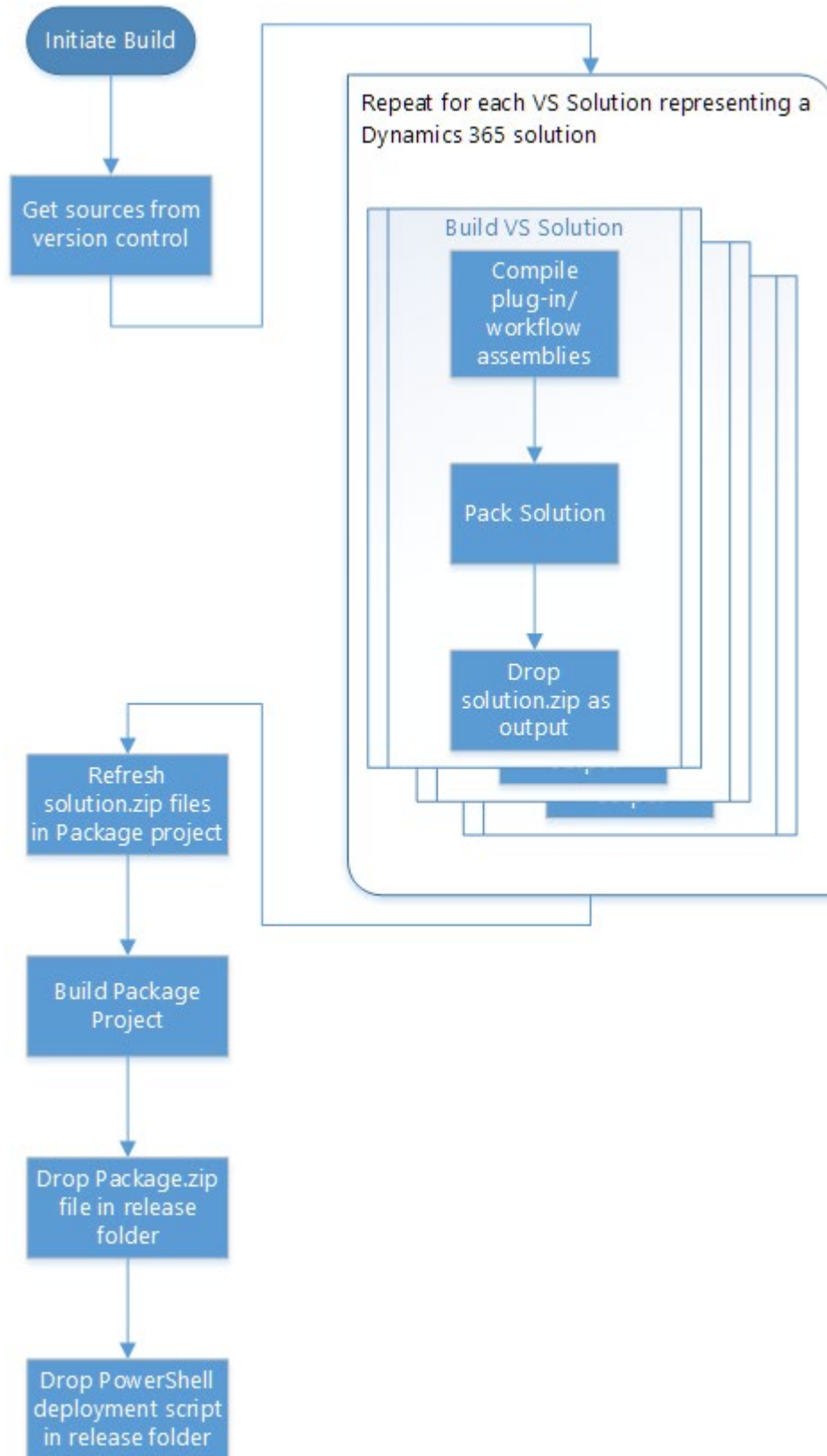


Figure 22 - Extending the build process to include packaging for deployment

UI Automation Test Library (Easy Repro)

Documentation Links:

<https://github.com/Microsoft/EasyRepro>

<https://blogs.msdn.microsoft.com/crm/2018/05/11/spring-2018-update-for-easy-repro/>

<https://www.nuget.org/packages/Dynamics365.UIAutomation.Api/>

Test automation can provide significant return on investment particularly for multi-phase or long-term implementations. One of the greatest challenges for functional testing through UI test automation is to identify the controls presented to the user. Typically, automation relies on control identifiers which can be impacted when platform enhancements are made to the user experience.

The UI Automation Test Library (a.k.a. Easy Repro), abstracts the test case implementation from the specifics of the UI controls. The API's provide an easy to use set of commands that make setting up UI testing quick and easy.

Tests are composed or extended within Visual Studio and can be run manually or as part of automated DevOps process. This offers a variety of uses from smoke testing a deployment through user acceptance testing. Tests may also be run on a schedule against production which provides additional confidence that behavior has not changed post platform updates.

Online Management API

Documentation Link: <https://docs.microsoft.com/en-us/dynamics365/customer-engagement/developer/online-management-api>

The ability to automate resetting to a known state is a key component of a repeatable and predictable Continuous Integration process. For Dynamics 365 for Customer Engagement apps, this was a gap prior to release of the Online Management API.

There are two approaches that can be taken to get to a known state:

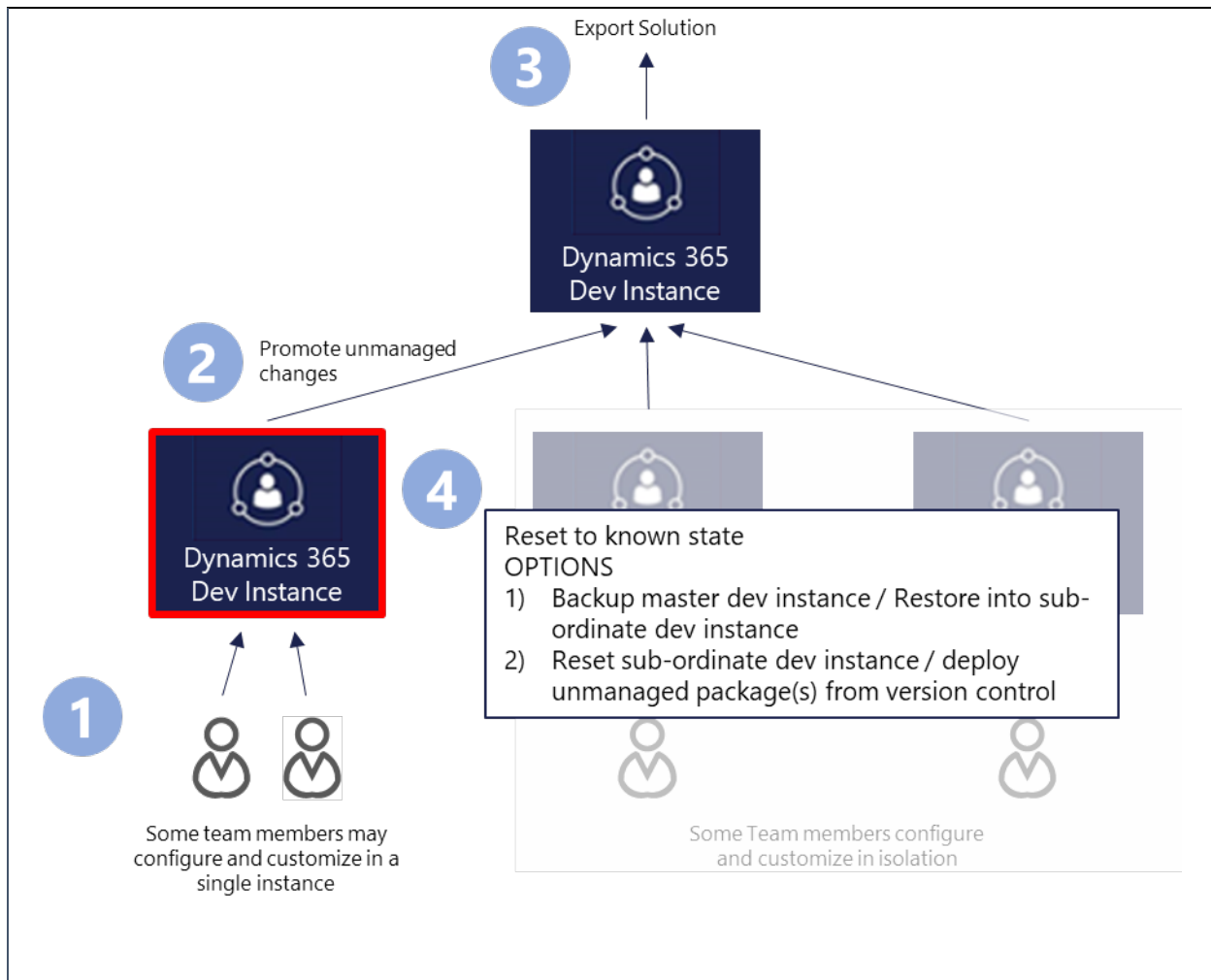
- 1) Reset instance and redeploy well known package version
more commonly adopted for CI as it is faster to reset than restore. A variant on this approach is to have a pool of free slots, create an instance, redeploy a well-known package and delete after completion of use
- 2) Restore a well-known backup version
Can be used for CI but for the reason above is less typically observed. This approach is highly useful where a copy of another environment is required. For example, to test deployment of V. Next on a copy of production.

The approaches taken will depend on the use case and number of instances available within the enterprise.

3rd party tools such as xRM CI Framework provide build tasks that wrap the online management API to simplify execution within build automation.

Table 6 - Approaches for resetting to a known state

Reset sub-ordinate development instances – single/multi workstream
Used to reset individual development or workstream instances and bring into consistency with the central or master development instance
Where a master or central development instance is used to compose unmanaged customizations across team members or workstreams, it is necessary to bring the sub-ordinate development instance into consistent state. This is achieved through the following approaches: <div><div>1) Taking a full copy of the master instance into the sub-ordinate instance (achieved via the Online Management API by backup of the master instance and restore into the sub-ordinate instance)</div><div>2) Resetting the sub-ordinate instance and deploying the unmanaged solutions or packages from version control</div></div>



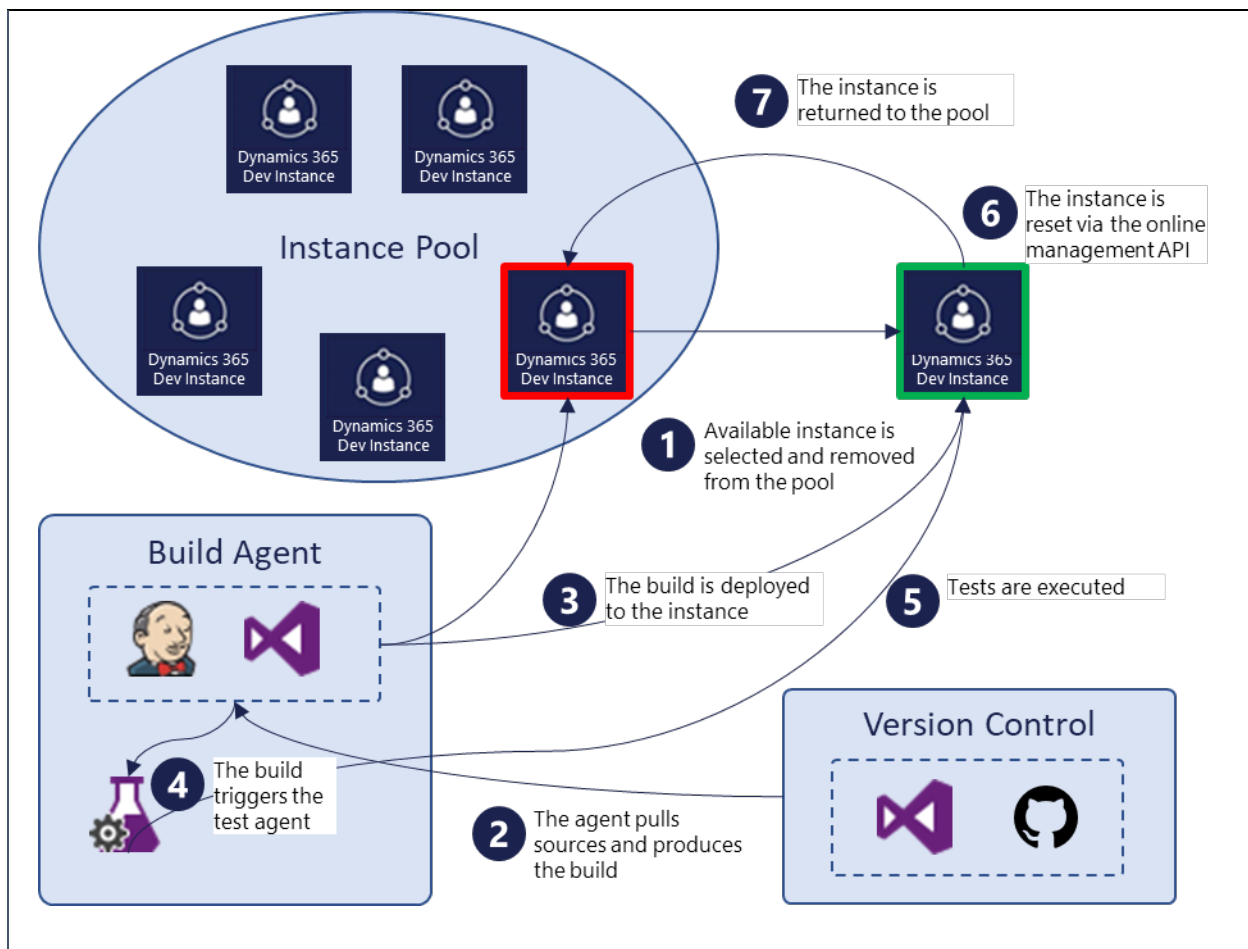
Provision CI test instances

Used to rapidly test deployment and basic quality as part of a gated check-in

Given the frequent nature of check-ins, a rapid approach to provisioning instances is required. It is likely that a pool of instances is required – at least one per build agent. These instances are partially provisioned – they are in a vanilla state or (V. current if the implementation work is delivering V. next)

A build is queued and initiates:

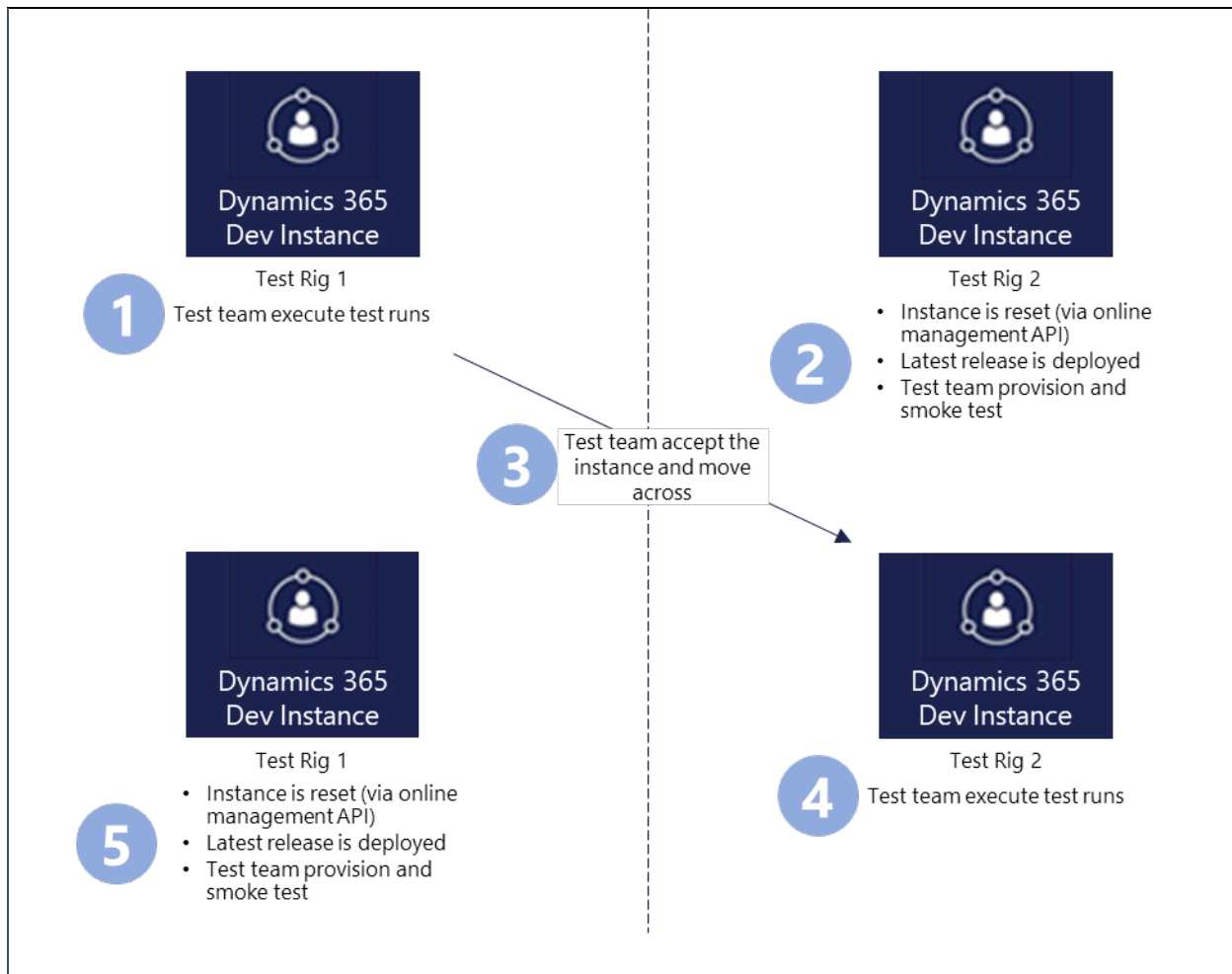
- 1) An available instance is selected
- 2) The build agent pulls sources from version control and produces the build
- 3) The solution or package is deployed to the instance
- 4) The build triggers the test agent
- 5) Automated CI tests are executed, and the gated check-in passes/fails
- 6) The instance is reset (re-provisioned if delivering V. next)
- 7) The instance is returned to the pool



Daily Build Flip-Flop test team instances

Used to enable daily testing efficiency (test on one instance while another instance is being provisioned with the next build)

Test teams often have multiple test instances to enable them to efficiently develop tests and execute test runs while in parallel the next build is being deployed. This ensures the test team are not blocked waiting for a build they can test. Once the new build is provisioned and accepted through smoke testing, the test team switch over to the instance and their previous instance is reset ready to take the next build. This is sometimes termed a flip-flop process.



Next wave deployment test/ Production deployment test

Used to verify a release can be deployed and will function correctly on top of a live implementation

This approach is essentially the same as the CI Test approach. For existing, live deployments that are being enhanced or patched, it is important that deployment and test verification are conducted in the manner they will be applied to production (i.e. on top of an existing configuration and code base).

Resetting the target instance is not enough in these circumstances. It is typical to follow one of two patterns:

- 1) Copy a source instance at production level into a target instance
In the simplest form, a copy of production may be taken into a sandbox instance however, it is common that production data is controlled sensitively. A typical alternative is to maintain a test environment at the same code/configuration level as production and to use that as the source for the copy.
The copy utilizes the online management API to automate the procedure of taking a backup of the source instance and restoring that backup into the target instance.

Following this provisioning step, the new version of the package is deployed, verified and test runs are executed.

2) Provision a target instance at production level

An alternative approach is to always provision from version control. With this pattern, the target instance is reset to “vanilla” via the online management API. The current live package version is retrieved from version control and deployed to the instance.

Following this provisioning step, the new version of the package is deployed, verified and test runs are executed.

Live Production deployment

Deploying V. next on top of V. current in production

By the time production deployment occurs, the automated processes are likely to have been executed multiple times against multiple environments. Issues with deployment predictability and reliability should have been resolved.

There are additional steps supported by the online management API that some enterprises adopt to safeguard production deployment

1) Creating a copy of production (achieved through backup and restore)

- For long deployments where users should not access the live system, this provides an option where the copy can be exposed in read-only mode to maintain a degree of operation to support critical live services
- In the event of catastrophic deployment failure (where production needs to be rolled back), there is an opportunity to consider demoting the production instance to a sandbox and promoting the original copy of production for faster recovery.

2) Put production into Admin mode

The online management API provides the ability to automate placing the production instance (or any instance) into admin mode. This prevents non-admin users accessing the instance and can be helpful during large deployment of large updates

Process and Automation Maturity

There is a broad spectrum of ALM needs when considering Dynamics 365 for Customer Engagement and standalone model-driven apps implementations. Application of good practice will vary based on the complexity and longevity of the implementation, the resources available and the anticipated return on investment.



It's important to appreciate the concepts and patterns covered within this whitepaper enable organizations to start anywhere on the spectrum and to plan how or whether they progress and mature their ALM practices further.

The following categorization aims to support the process of determining the needs for a specific implementation and the typical order of investment to evolve. Not all implementations will evolve to adopting all principals or application thereof. At a minimum, the implementation should satisfy the [ALM self-assessment checklist](#)

Table 7 - Process and Automation Maturity Matrix

Solution Management			
<ul style="list-style-type: none">•Single solution approach•Manual Export / Import•Basic version control (file system / solution.zip held in version control system)	<ul style="list-style-type: none">•One or more solutions•Well defined solution boundaries and solution dependencies•Manual extraction (unpack) of solution via Solution Packager and check-in into version control system	<ul style="list-style-type: none">•Solution segmentation is practiced•Solution Patching is practiced•Solution export and extraction (unpack) via Solution Packager is automated	<ul style="list-style-type: none">•Re-useable common solution libraries are defined and refactored
Level 1	Level 2	Level 3	Level 4
Build Management			

<ul style="list-style-type: none"> •Manual build for custom code components •Manual refresh of plug-ins/custom workflow activities within solution •Release management/build quality maintained through file system 	<ul style="list-style-type: none"> •Basic build automation •Plug-ins/custom workflow activities are compiled and mapped into solution •Solution is packed via the Solution Packager •Solution is “dropped” as a release 	<ul style="list-style-type: none"> •Check-ins invoke gated check-in process and CI build •Enhanced build automation •Package(s) refreshed, and Package Definitions built •Package definition(s) “dropped” as release 	<ul style="list-style-type: none"> •Advanced build automation •Build extended to invoke deployment via Package Deployer •Build extended to invoke test execution •Build tagged with quality factor for Release Management
Level 1	Level 2	Level 3	Level 4
Test Management			
<ul style="list-style-type: none"> •Manual 	<ul style="list-style-type: none"> •Manual traceability back to requirement 	<ul style="list-style-type: none"> •Automated tests defined and invoked manually 	<ul style="list-style-type: none"> •Test execution triggered automatically as part of build, deployment or service monitoring •Automated traceability of tests from requirement through bug creation
Level 1	Level 2	Level 3	Level 4
Deployment Management			
<ul style="list-style-type: none"> •Manual deployments steps are clearly documented and repeatable 	<ul style="list-style-type: none"> •Solution(s) deployed through script 	<ul style="list-style-type: none"> •Packages (solutions and configuration data) deployed via Package Deployer through script 	<ul style="list-style-type: none"> •Instances managed through automation via Online Management API •Releases deployed to downstream environments from Release Management tool based on quality
Level 1	Level 2	Level 3	Level 4

Where should investment be made?

There is no right or wrong answer but typically, moving up one level across all dimensions is likely to produce better incremental value than investing heavily in a single dimension. In some cases, no value will be derived by moving up a level or more in a single dimension given there are interdependencies across the dimensions.

Below are some example considerations that have been seen to drive the decision-making process within enterprise organizations

Deployment to downstream environments

Most enterprises have a clear separation of duty between development and operations. It is uncommon for developers to have direct access to environments downstream of development. This results in a need for a member of operations to execute deployments for pre-production and production environments.

Repeatable and well documented manual processes can be used effectively but they typically require more time resulting in operational inefficiencies. There is also the possibility of introducing human error and therefore, automating deployment of the solutions and associated reference data can provide a faster and more predictable deployment. The effort involved is relatively small resulting in good return on investment.

Note: Automated package deployment via the package deployer can be achieved even where there is a manual solution packaging and build process.

Solution Management Version Control

Basic version control is a pre-requisite to build automation. There is a significant increase in value through version controlling the extracted (unpacked) form of exported solutions. The additional granularity significantly improves traceability of change which can help to address issues and identify how and where they were introduced.

Granular changesets are constructed through the Solution Packager tool which can be executed manually through the command line or included into a scripted process. The cost of introducing granular version control is relatively low, it is also the gateway to maturing other dimensions of process and automation. As such, this is a strong candidate to consider for good return on investment.

Build Management

During his time at Microsoft, Jim McCarthy wrote an article published under the title "21 Rules of Thumb for Shipping Great Software on Time". To quote rule 11 from that article,

"If you build it, it will ship. Conversely, if you don't, it won't".

While this rule was stated over a decade ago it has never been more relevant than in the current cloud and DevOps era.

Basic build management does not need to incur significant resource effort, particularly when compared to the value derived. A build process should be executed at least daily and provides the heartbeat of the implementation. The time it will save and the defects it will prevent, offers an opportunity to continue maturing the capability throughout the implementation lifecycle. Given the frequency the build will be invoked, it is easy to extrapolate the return on investment that will be derived by avoiding blockers to deployment, testing and acceptance. It is never too late to commence build automation – even if enhancing past a basic level will not be required.

Test Automation

Undoubtedly, all enterprises understand the value and need for good test execution practices. In terms of maturity levels, the same is not always reflected. Often, at least one form of testing within the implementation lifecycle (unit testing, functional testing, integration testing, user acceptance) is conducted through a time intensive manual approach. This introduces a human error factor that can result in issues being missed or areas of functionality going untested due to capacity or time constraints.

Test automation is sometimes seen as being costly to invest in both in terms of complexity and effort to maintain. This is particularly true when the implementation is already in production and a subsequent phase is being delivered. Like build automation, once written, automated tests pay back their cost through every execution. Each defect they prevent being deployed to production is likely to save person-hours of effort to identify, diagnose and resolve. This provides two options:

- 1) A more efficient test cycle (reduced duration and /or reduced resource requirement)
- 2) An opportunity to invest the person-hours saved in further automated test coverage

The availability of the UI Automation Test Library (Easy Repro) is one option to reduce the barrier to entry for test automation. Whatever test framework is the tool of choice, test automation arguably has one of the highest returns for a moderate continued investment.