SAP Development Approach

Purpose

The SAP Development Approach document will be used as a guide for all SAP development activities undertaken in S/4HANA, BTP and Reporting.

The document includes details of the principles expected to be applied by all development teams and decision trees to facilitate the solutioning based on best practice.

This document will be used in collaboration with Development Standard & Guideline documents that contain the detailed rules and naming standards as well as further details on best practices.

Objectives

The objectives of this document are to:

- Define the principles and rules for SAP software development at Syensqo
- Provide an unambiguous set of modern development practices in an SAP environment.
- Set the expectations for code quality.

Development Practices

Behaviors and Expectations

It is the expectation at Syensqo that a level of software development professionalism is maintained by all development teams engaged in development activities, regardless of the size, scope or system. The following section will outline techniques and practices that all teams are expected to follow to ensure that development at Syensqo achieves a high level of quality, robustness and user acceptance.

Behaviors

- Development teams will report progress accurately.

- Development teams will strive to build quality software by utilising the current tools, approaches and frameworks provided by the platform. Doing things, the “old way” simply due to limited understanding, ease or comfort is not an acceptable justification**.**
- Development teams will ask business focused questions as they arise to ensure that requirements are met.
- Developers will work closely with Functional SMEs and Testers to build acceptance tests for implemented features.

Expectations

- Syensqo can set development priorities within a phase or iteration.
- Development teams can expect to have a clear indication of any development priorities requested by Syensqo.
- Development teams will be empowered thorough the provision of the right tools, relevant system access and the availability of the right people.

Code for Maintainability

Over time a piece of code will have numerous maintainers and undergo minor and major changes. Each change will affect the architecture of the software in some way, as well as reflect something of the developer that made the change. Left unchecked, this process will result in a code base that will become more difficult and costly to enhance and maintain over time. This is commonly referred to as the accumulation of technical debt.

A key expectation for developers at Syensqo is to write maintainable code. The following well established practices will guide the way code is written, modularized, organized into packages, and then assembled into working systems. By following these practices code maintainability will be assured.

Rule	Code for maintainability.

Separation of concerns

Separation of concerns is the practice of decomposing software into distinct areas that represent a specific task or behavior. The resulting concern will change for the same reason, meaning if a change to a behavior is required, the change will be localized to the one area of the code.

Separation of concerns is realized using language modularization techniques and code organization (assigning code to packages or repositories).

Separation of concerns does not mean putting a single concern into one class, rather using appropriate modularization techniques to group relevant artefacts together.

Good separation of concerns results in the following:

- Stable code – changes are localized rather than scattered resulting in simpler testing and less errors
- Code is easier to understand – by only having one concern the code is clearer
- Code is easier to test – testing can focus on a single concern

Rule	Follow the principle of separation of concerns.

Abstraction

Abstraction manages complexity by allowing a developer to work at a level of detail that is relevant for the current context. For example, it is easier to work with the concept of a *Web Page*, rather than a long HTML string, or the concept of a repository rather than DB tables and SQL statements. By using abstraction, developers will build models of the problem domain which will guide the implementation approach.

Proper use of abstraction will allow components to be reused to solve different use cases. As abstraction will often include terms from the business domain, this approach will assist during design as the developers and the business will speak the same language.

Rule	Use abstraction to manage complexity.

Code Organization

All modern languages offer a code organizational capability. Without a code organizational structure in place code is added without thought to its purpose causing the code base to deteriorate quickly.

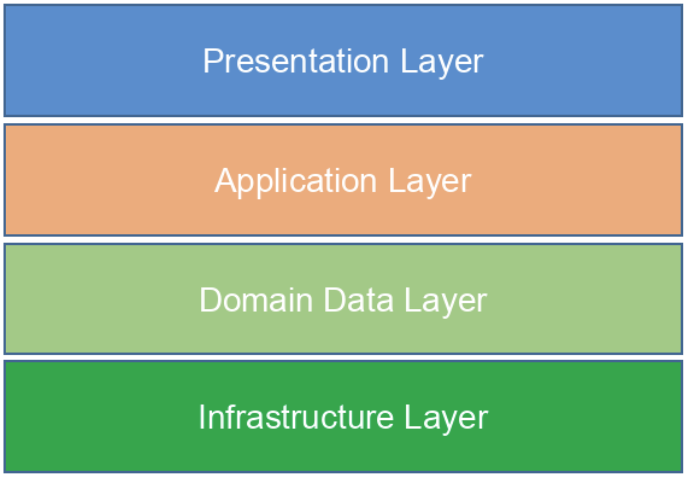
A well organized code base avoids this outcome by providing a level of architecture to guide development.

Syensqo has decomposed its business functions into 4 levels and this decomposition will be used for code organization for all development activities. This aligns with the goal of writing code that observes the principle of s*eparation of concerns* as all code artefact will contribute to a solution for a specific functional process. As a rule, each business function can be composed into larger Business Scenarios which cross different parts of the organization. This also applies to the code artefacts – code artefacts from different functional processes can be composed together to realize new scenarios.

Within each functional process, code artefacts are separated again based on architectural layers. This explicitly enforces a technical separation of concerns within each process.

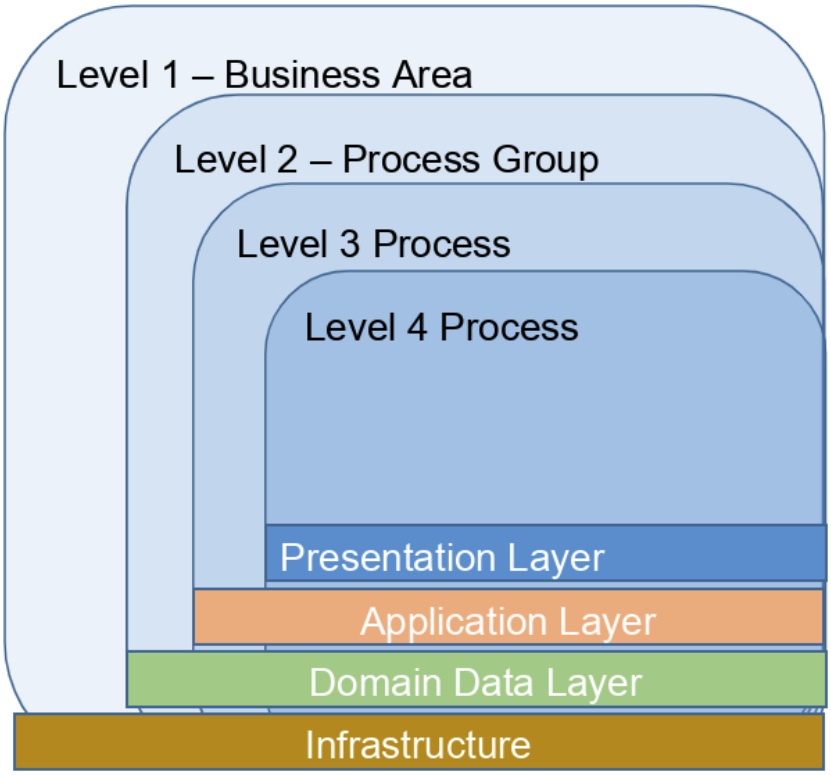
A stereotypical layering approach is as follows.

- Presentation Layer – artefacts are responsible for presenting information to the user. For example, SAPUI5 applications.
- Application Layer – artefacts expose application logic as released APIs. For example, the use case “Calculate the cost of the Purchase Order” would be found in this layer. Application layer artefacts orchestrate objects in lower layers to achieve this.
- Domain Data Layer – artefacts represent the required domain data model or data objects, for example, Account, Person, Invoice or Company on the database level.
- Infrastructure – artefacts represent the foundational layer that supports the Domain Data, Application and Presentation layers.



Layers apply dependencies from the top down. From the diagram above, the Presentation layer depends on all other layers, while the Infrastructure layer has no immediate dependencies. The order of the layers shown above is what is commonly used.

The diagram below illustrates how the functional decomposition and architectural layers will work together. Although aligning with the functional decomposition model and using architectural layers is not optional, it is not mandatory to use all layers. For example, a SAPUI5 application may be aligned with a Level 4 Process and only requires a Presentation Layer for its artefacts to be developed, or the Data Layer applies on Process Group level. Infrastructure is the foundational layer.



By ensuring code is organized in this manner, practices such as the *separation of concerns* can be re-enforced.

Rule	Use Code Organization to emphasize application structure.

Single Responsibility Principle

Whereas the *separation of concerns* describes how code should be broken into distinct areas, the Single Responsibility Principle (SRP) defines an approach for determining what each code artefact should do. The single responsibility principle is generally defined as “a class should have only one reason to change”. This principle also helps decide if a new feature should be added to an existing artefact, or if a new artefact should be created.

When describing an artefact that follows the Single Responsibility Principle there should be no need to use conjunctions like ‘and’ or ‘or’; this is usually an indicator that the artefact has multiple responsibilities.

By following this principle artefacts won’t become collections of unrelated functionality and data, instead artefacts will remain clean, allowing simpler maintenance, enhancement and reuse. Artefacts that observe the Single Responsibility Principle are said to be cohesive, and this can be measured as part of code quality reporting using the SAP Code Inspector. Metrics such as Class and Method size, cyclomatic complexity, coupling and cohesion will be continuously collected.

Rule	Code must follow the Single Responsibility Principle.

Code for Readability

Characteristics such as formatting, meaningful names and appropriate comments will determine how readable the code will be. Code readability is a critical attribute for maintainable code.

Naming

Names need to convey to the reader the purpose of the artefact, variable or function. The name must be unambiguous in meaning; it must do what it says it does.

The following rules support good naming practices.

1. Always choose a meaningful name. A meaningful name shouldn’t need to be clarified with a comment
2. Avoid abbreviation whenever possible. Commonly used abbreviations such as HTML and XML are acceptable. If abbreviation is needed, abbreviate consistently and use standard abbreviations where they exist.
3. Don’t encode scope information into variables. Specific language encoding is covered later in this document.
4. Choose a name that is relevant for the context it is being used in.
5. Use singular or plural versions of the name to communicate the arity.
6. Avoid words that have special meaning in the implementation language as they may implicitly imply meaning that is not intended – for example, the name setOfCustomers may imply to a Java developer that the Set interface is used in the implementation. In this case, customers is a better name.
7. Names such as customer1, customer2 or theCustomer should never appear in the same scope. If the variables are needed then they should be named for their purpose – if they are all customers, only one variable is required!
8. Code is discussed during design, peer review and support. Use names that can be pronounced to make this easier.
9. Keep in mind that good names will make source code searches simpler.
10. Use nouns or noun phrases for classes, types and variables. Use verbs or verb phrases for methods, functions and events.
11. Be consistent when naming for the same concept. For example, don’t use query, find and request when referring to data access methods – choose one.
12. Names should be changed when better names are found or the intention of the named object changes.

The following is a guide for creating an abbreviation for a name when no common abbreviation exists.

1. Omit the end of the name. *Abbreviation* becomes Abb.
2. Omit all vowels from the name. If the first letter of the name is a vowel this must be kept. If the word starts with a double vowel both should be kept. If further abbreviation is required, replace double consonants with a single consonant. *Abbreviation* becomes Abbrvtn (or Abrvtn)
3. Create an acronym. *Accounts Payable* becomes AP.

Rule	Naming must be clear, context- and intent-based.

Comments

If not well maintained and used appropriately comments become noise in the code, detracting from what is important. All comments will add a maintenance overhead to source code. That said, comments form an important part for readability in software development.

Code is always the source of truth for what software does. Comments will only reflect this if they are maintained continuously. For this reason, comments should be used carefully. Comments should be used only when they add to the readability of the code. Writing comments for the sake of writing comments can adversely affect the readability of the code.

Comments provide a mechanism for the developer to explain what the code is doing. This is usually a heuristic indicating that the code needs to be refactored into something better.

Rule	Code must be refactored if the intention of the code is unclear before a clarifying comment is added.

The following rules should be applied when adding comments

1. Don't add redundant comments that repeat what the source code is saying
2. Don't use boilerplate header comments. These provide no new information and go out of date quickly
3. Don't use comments to track changes to source code
4. Don't use comments as an alternative to code deletion
5. Use comments for legal information such as copyright if required
6. Use comments for interesting design decisions that may be relevant to future developers
7. Use comments to clarify the intent if it cannot be clearly expressed in code
8. Comment as close to the relevant source code position as possible
9. Use comments to highlight possible consequences of a change
10. Use comments to escalate the importance of a piece of code

Language

Always use English for naming and comments within source code and for development artefacts. Most modern programming languages use English for their syntax so writing code in another language will the affect the readability and maintainability of the code.

Rule	English is the language to be used for all developments.

Robust Software

Robust software expects and manages failure. At a technical level this means understanding where failure can occur and providing a mechanism to handle this. When software components are composed together to build systems a chain of possible failures is created. These need to be understood and mitigated.

The possible impact of a failure (or category of failure) should be used to determine how it is handled. For example, a failure that causes data loss is of much higher importance than a failure to load the weather forecast widget in the home page. Of course, failure to load the widget should not impact the rest of the home page. Once the impact of failure is understood, an appropriate handler can be implemented

Software handles failure through exceptions. These provide the opportunity to alter the execution of a program to manage a failure when it is detected. Software should fail in a controlled manner, providing context appropriate information to the user. A user should NEVER see a stack trace or a low-level exception.

At a more sophisticated level, patterns of exceptions can be handled in pre-determined ways. For example, in the case of the weather widget, after a pre-determined number of retries the system may determine that it is permanently unavailable and not attempt to load it again for an hour. This saves system resources and is an acceptable response in this context. This pattern is called a "*circuit breaker*".

When writing code for handling exceptions, the following rules should be used.

1. Always use a suitable exception. The exception that is raised must reflect the error that occurred and provide enough context to be handled correctly.
2. Exceptions should be caught and handled at the time or allowed to propagate to a higher level in the call stack.
3. When propagating exceptions don't cross architectural boundaries, instead catch the exception and raise a new exception that is relevant for the higher architectural layer. If supported by the language, attach the lower layer exception to the new exception to preserve the failure chain. For example, an exception raised in the database layer should not be passed directly to the application layer – instead capture the database exception and raise an exception relevant to the application context.
4. When propagating exceptions always leave the current level in a known consistent state. For example, close any open files and release any database locks.
5. If an exception can be prevented using appropriate pre-conditions (such as validating upper or lower bounds) then an unchecked or dynamic exception should be preferred. When an exception cannot be reasonably prevented use checked or static exceptions.

As a rule, only one exception block (for example try/catch) should appear in each method or function. This practice enforces a *separation of concerns* by handling the success or failure of a single concept.

Rule	Each method or function should contain only one exception block.

Message Logging

Logging should be used as appropriate in software development. In productive environments where debugging is difficult or forbidden, logging will provide information on what went wrong and what was happening before the issue occurred.

For logging to be useful, it requires the following characteristics.

1. Logs should be human readable.
2. Logs should be stored as strings allowing easy searching

3. Each event should be clearly timestamped
4. Use a unique identifier to allow correlation of related events
5. Use standard categories for log events – Information, Warning, Success, Error and Debug
6. Always include the source of the log event

Each application environment provides its own logging tools and frameworks, these should be used.

Custom logging frameworks should only be written in exceptional circumstances and require the approval of the Syensqo Technical Architect.

It is important to distinguish between the technical logging and tracing used by developers for debugging, and functional logging which provides information to the end user. The ability to switch logging and tracing tools on and off and adjust the trace level should be considered. Finally, log retention periods and log archiving strategies must be understood for each framework.

Testing

Good software is delivered with tests. The process of developing complex custom software at Syensqo will include what is known as “technical facing tests”. These tests are typically written in the same language as the software itself. Developers should deliver both Unit and Component tests to demonstrate the correctness of the delivered software as part of the quality process.

Iterative development requires that technical facing tests are executed frequently. This practice closes the feedback loop on the change the developer has made and alerts the developer to any bugs that may have been introduced by the change.

Tests also guide the developer towards the solution. By utilizing test first or test-driven development a developer first describes the required functionality as a test and then implements code to make the test pass.

Unit and Component tests should be automated. Language specific tools can be used for component testing if needed.

Testing at the service layer (use cases/features) and above falls into the area of “business facing tests” and this is not in scope for this document.

Because unit and component tests are kept up to date as the system is changed, they are a critical part of the documentation of the system. Tests describe how the components, methods and functions behave and how to interact with them. Tests are the “tutorials” for new developers and support teams.

Rule	Software will be delivered with Unit and Component tests.

Application Security

A user’s job or role will determine if a specific application can be used. Application level security is then applied to determine what a user can do with the application. For example, can a user only display Invoices, or can they create or change them. This level of security will affect what controls are presented to the user (menus and buttons), what state a field is displayed in (read only) or if a field is displayed at all (hidden).

Each platform provides access control capabilities that can be used to determine a user’s authorisation. Based on this information, the state of the application as presented to the user is modified programmatically as required.

The application security model would be assessed on a per application basis and implemented as required using platform specific tools.

Rule	Authorization group will be maintained, SAP pre-delivered &NC& will not be used.

Rule	Custom code must always be executed in the context of an authorised user, i.e. after an authorisation check has been performed by the system. For code directly invoked by a user (e.g. programs, transactions, Web Dynpro, Fiori, also web services, Odata, etc.), an explicit authorisation check must be performed before business or data access logic is executed. Relying purely on Start authorisation is not enough to meet these criteria. Custom code executed within an already-authorized context (e.g. BadIs, user exits, enhancement spot, etc), is exempted from this requirement because the check would have been performed by the standard SAP code.

Re-use

Developers should minimize duplication through the extraction of common functionality into functions and methods. Duplication of code increases the risk associated with change as fixing an issue in one place may not guarantee that the same issue doesn’t also exist elsewhere in the code. As a rule, the third time something is repeated, it should be refactored into a re-usable component.

There is risk in re-use. When a function is extracted, an abstraction is created. This abstraction may prove to be incorrect for later re-use, encouraging the extracted code to compensate through conditionals. This breaks the single responsibility principle and impacts the quality and maintainability of the code.

When re-using existing code ensure the abstraction is right. If not, duplication appropriate to the context is the best approach – favor simplicity before generality if both options are equally feasible.

Patterns

Software Patterns provide generic solutions to well-known problems in software development. A pattern provides an approach that can be contextualised to a specific implementation. Software patterns provide a common language for developers to use; there is no ambiguity, and new team members don't require problem domain knowledge to understand what the pattern does.

Patterns, by applying a known approach, drive software architecture away from haphazard collections of classes and functions towards a structured, maintainable solution. For example, the Model View Controller pattern is used to ensure a clear Separation of Concerns in UI development while the Strategy pattern is used to allow application logic to be dynamically substituted at runtime.

Rule	Use patterns where appropriate.

User Interface Framework

Syensqo has invested in a diverse SAP platform which spans both hosted ABAP based applications for delivering core business suite functionality and cloud-based applications such as Success Factors for HR, C4C for CRM and Concur for Travel Management. Delivery of these applications will be primarily via a Web Browser, and additional UX concerns such as responsive design for Mobile and Tablet users' needs to be considered when developing custom user interfaces or extending existing applications.

To support the development and extension of these new applications SAP has delivered the *UI Development Toolkit for HTML5 (SAPUI5)* which is a UI framework based on Jquery, HTML5 and CSS3. This framework is designed to provide a modern web experience for consumer grade applications and casual enterprise users by enabling the development of highly interactive and focused applications that support responsive design. SAPUI5 applications can be targeted at desktop, tablet and mobile phone users.

SAPUI5 apps are compliant with the SAP UX Strategy and deliver on the Fiori UX experience. The Fiori Elements framework allows you to generate UI5 apps based on meta data additions in CDS views and RAP. If the required app is targeted to expert users, or has a higher complexity than Fiori Elements can handle, a Freestyle app needs to be developed. Freestyle apps start with a blank canvas. These apps can be built in SAPUI5 or React. All SAPUI5 apps can be deployed on either S/4Hana or BTP. React apps can only be deployed to BTP.

ABAP Dynpro is not allowed in custom development. If there is a legitimate requirement to transform an existing Dynpro application to conform to the Fiori UX experience, i.e. for a mobile device screen, or to simplify the UI and a suitable standard SAPUI5 cannot be used, then SAP Screen Personas can be used to transform the Dynpro to a Fiori type User Interface.

SAP has positioned the Business Technology Platform (BTP) as the extension platform for all SAP Solutions. When building front end extensions using the BTP, the SAPUI5 framework or React JS is to be used.

The following table illustrates the preferred UI framework for custom developments for several possible use cases.

Use Case	Dynpro/ Floor Plan Manager WDA	Fiori Elements	Freestyle (SAP UI5/React JS)
Single purpose focused application or casual use	✗	✓	✗
Transactional application targeted at an expert user	✗	✓	✓
Responsive Design; Support for tablet or mobile phone.	✗	✓	✓

Rule	Choose the appropriate UI framework based on the use case.

Reporting

Realtime analytics will be restricted to small data sets so as to not impact performance on the source system.

This is typical of generic analysis of financial data vs plan for say the current month for a company. The same could be applied to the material ledger.

In these scenarios, you would want to apply the restriction to the dataset via parameters or filters in the CDS view.

Besides small data sets, we also want to avoid cases where data sets are accessed frequently. The concept is to cater for ad-hoc requests rather than be a frequently recurring path to access data.

The CDS views are using associations, which means that master data is only accessed on demand. This does result in an efficient data access request.

Other SAP development tools to be considered where appropriate:

- Custom Analytical Queries (F1572) - however, this uses deprecated views rather than projections
- Manage KPI's and Reports (F2814)

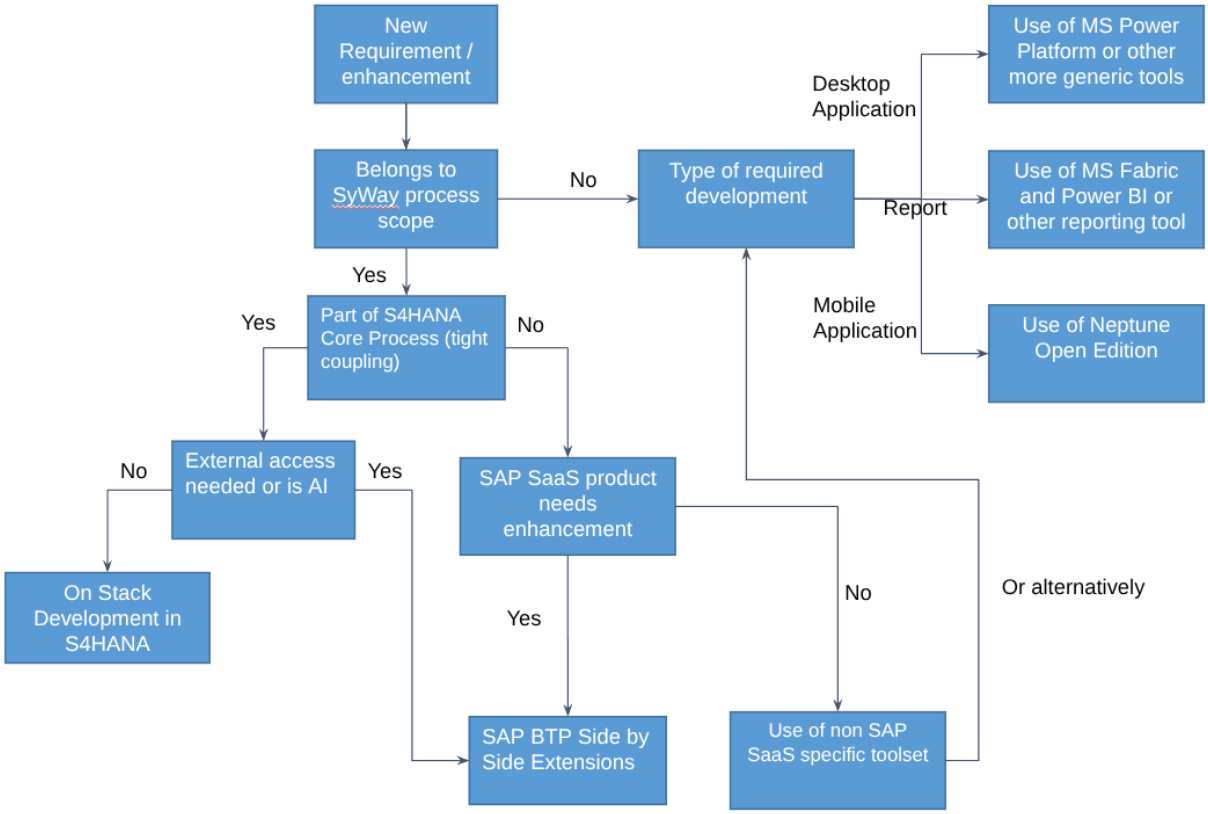
Extraction of data to be acquired in Datasphere will require a delta scenario by way of Change Data Capture (CDC) being applied to the CDS view. In this case you would not want to have parameters or filters. If no fields available for a timestamp, then a pseudo-delta can be used where for example the current month can be extracted in full.

Rule	Realtime analytics only to be used for small data sets that are accessed on an ad-hoc basis.

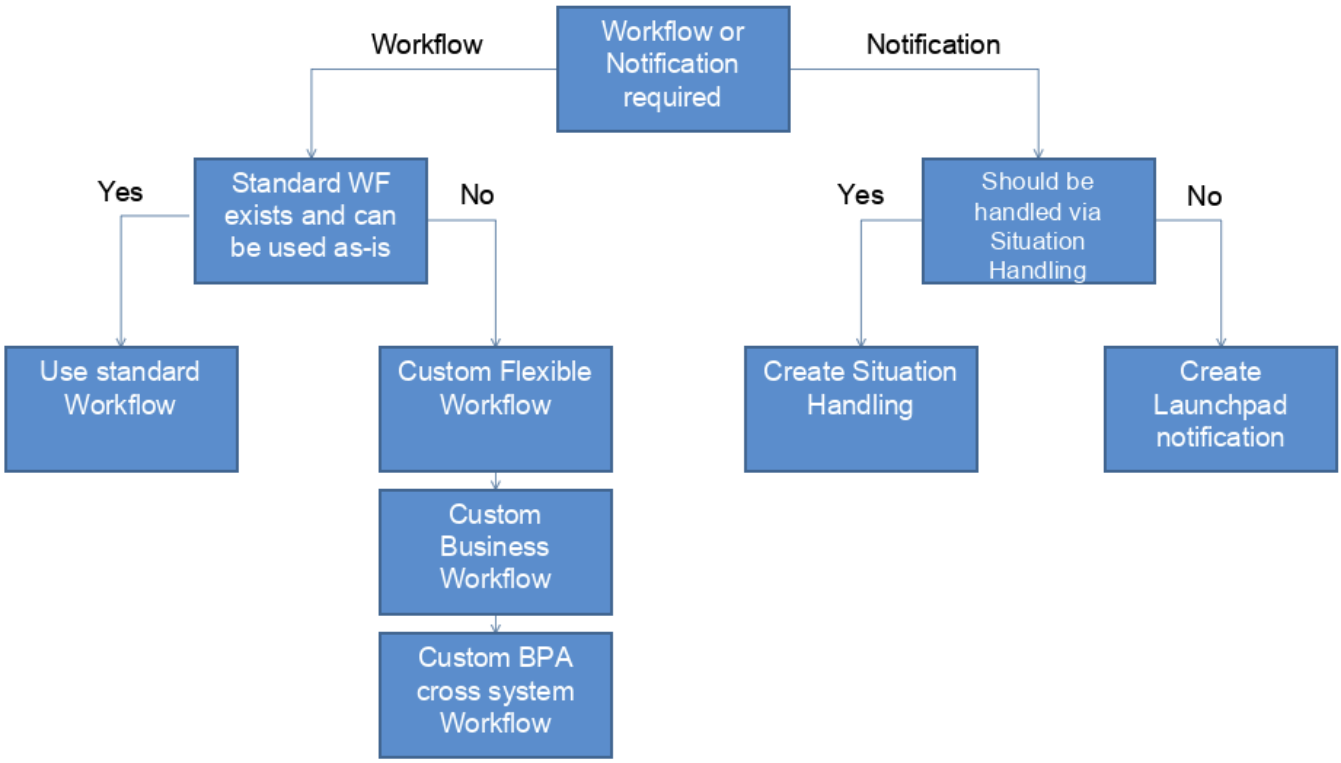
Development Tool Decision Trees

The following set of decision trees shows the decision path to take to find the right development tool and approach to deliver compliant objects in SAP.

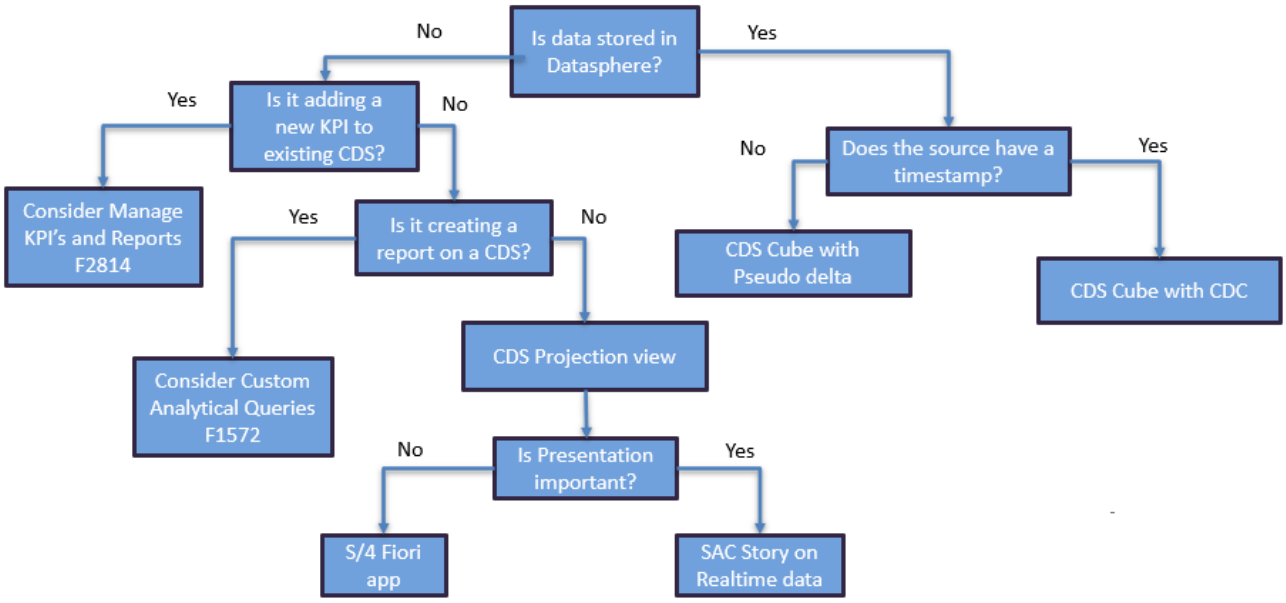
BTP vs S/4HANA on stack vs Others



Workflow

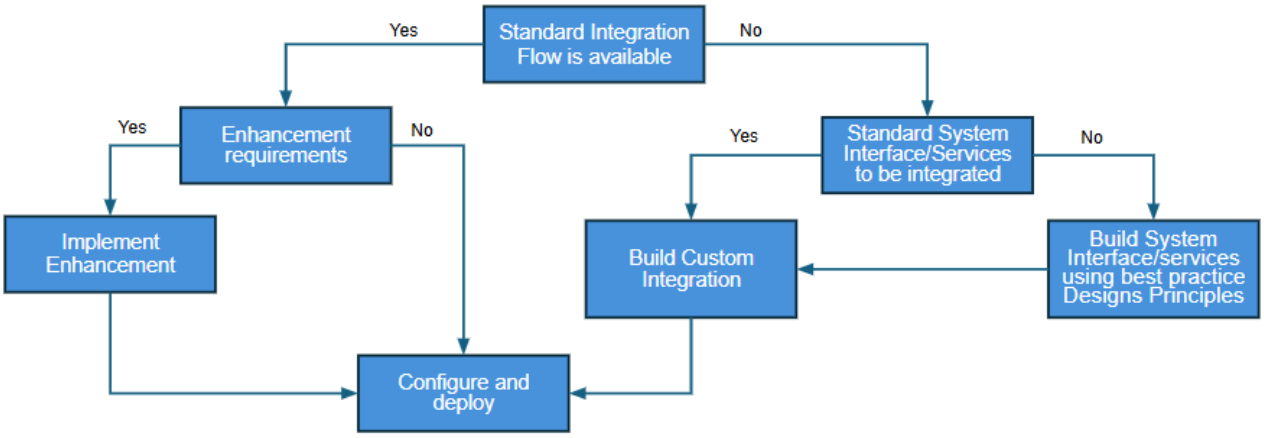


Analytics

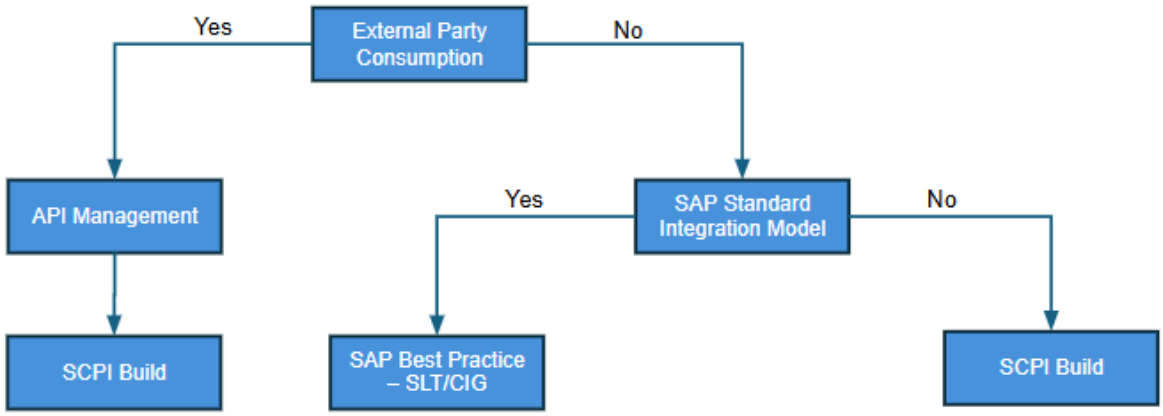


Integration Process

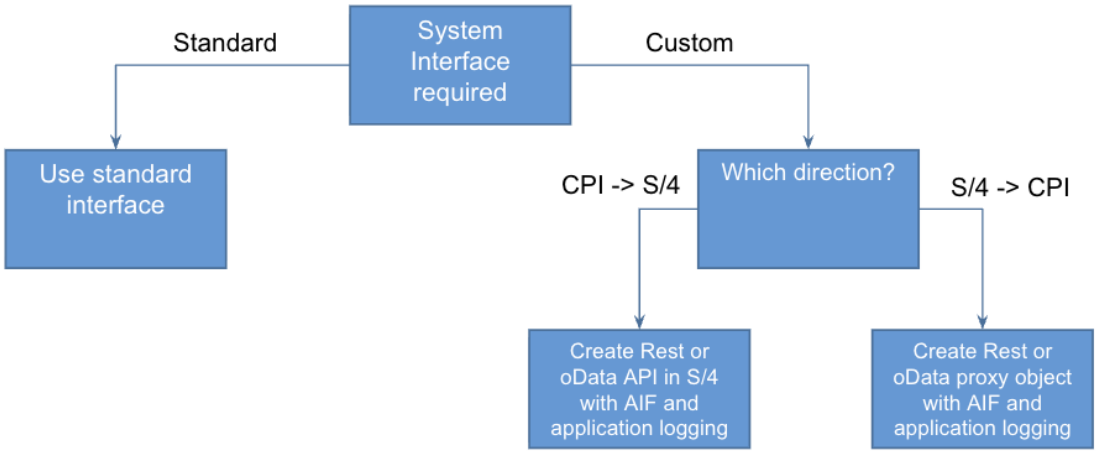
SCPI - Integration Development Process



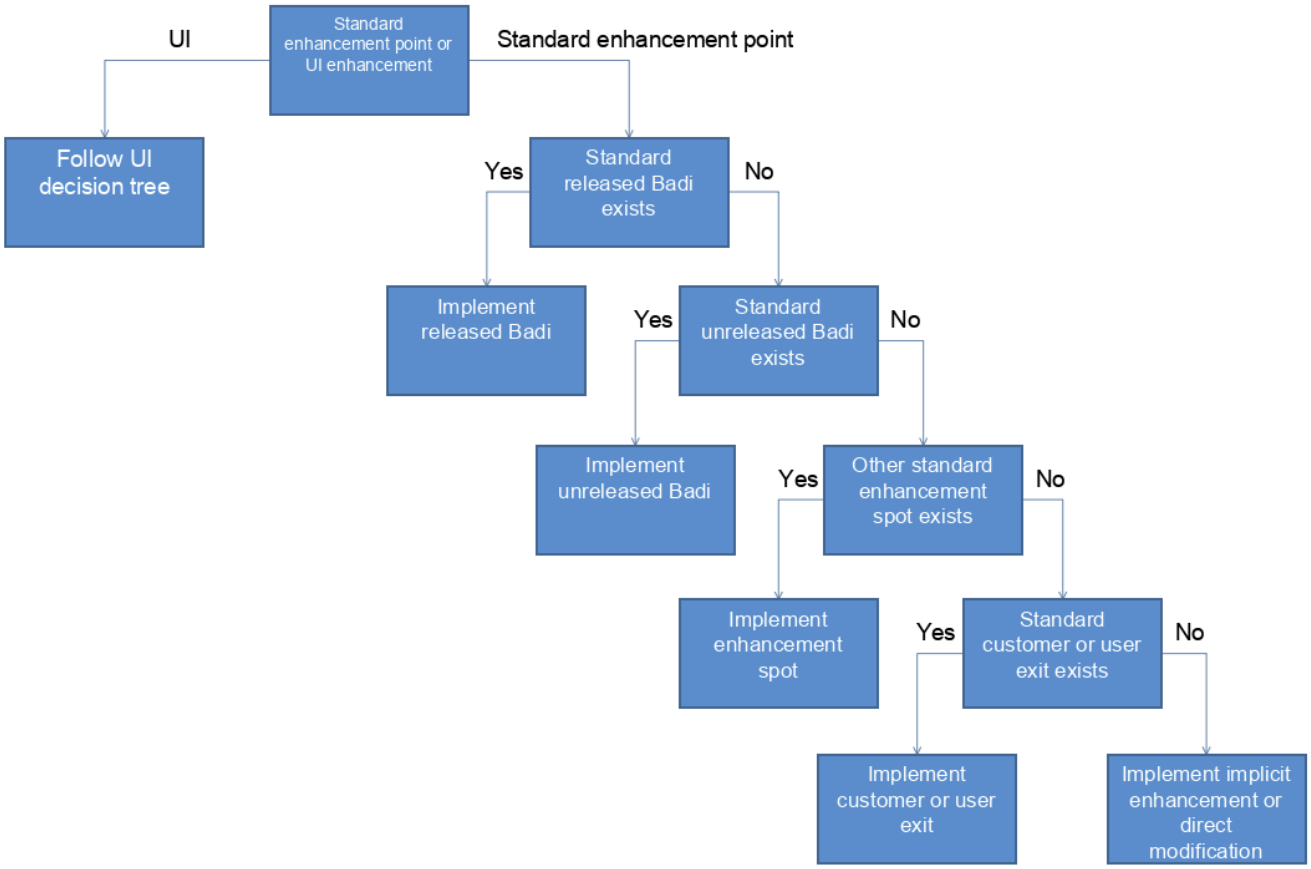
APIM Management/CI/Event Mesh



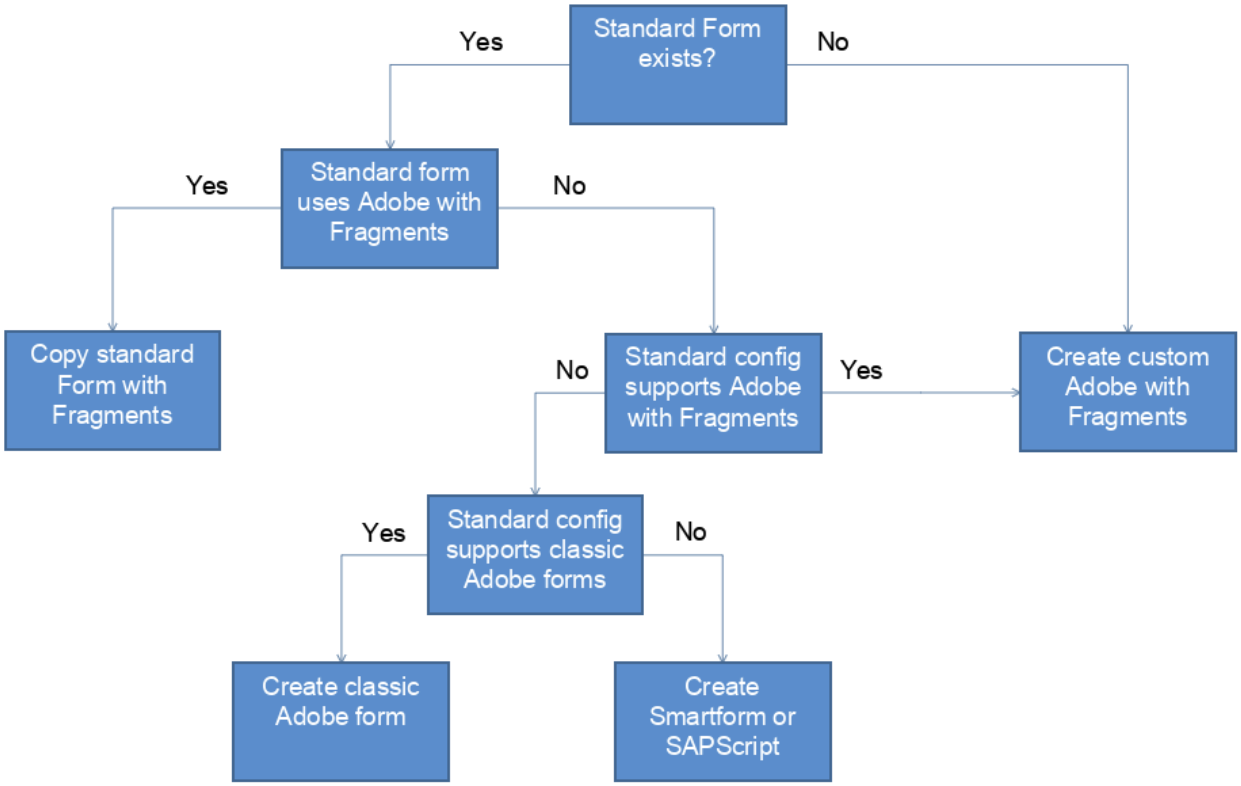
System Interface



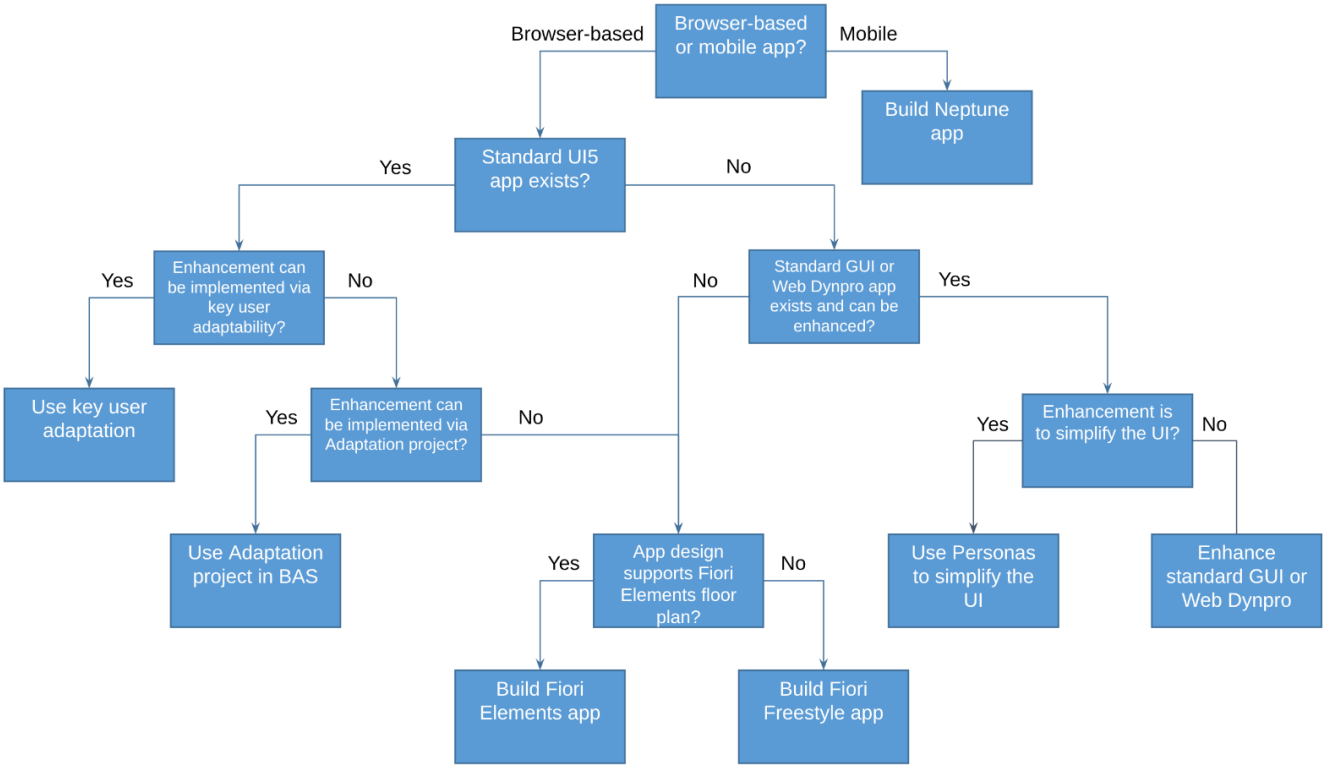
Enhancement



Form (Output)



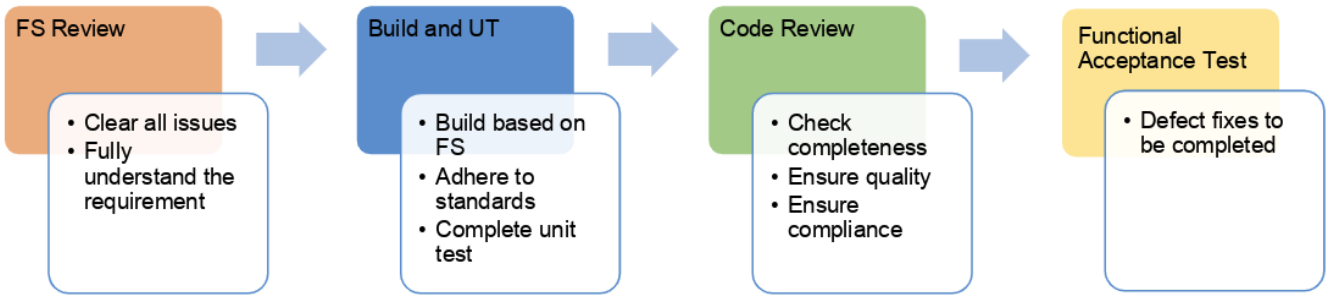
User Interface



Link to the original file containing the drawings: SAP Development Approach drawings

Development Process

Every development requirement must go through four phases as shown in below picture. Failure to comply results in a “no-go” decision at go-live.



The first review check point is the FS review that is required to be conducted once the FD has been peer reviewed within the functional area and submitted for development review. During the FD review it is vital that the reviewer fully understands the requirement and clears the document to start the build. All issues need to be logged, updated and closed in an issue log.

After the FS has been reviewed and approved the actual development can start. Only what is in the FS will be build. Additional requirements or changes in tier need to be updated in the FS and re-reviewed before build continues.

The second review check point is the Code review. It is vital that the development is reviewed once UT has been completed. Development completeness and adherence to standards must be reviewed to ensure high quality.

At the time of Code review, all documentation and checklists must be maintained in the Jira Custom Development card.

Last phase is the Functional Acceptance Test (FAT). The development will be tested by the functional team to validate that the required scope has been developed. Defects during FAT need to be fixed in a timely manner.

Further details of expected review scope are in the following sections.

FD Review

- FD review can only be considered completed when there are no open issues anymore
- Requirement changes during Build need to be added to the FD and re-review is required

Code Review

- Code Review to be requested after UT
- Code review can only be considered completed when the UT has been completed
- Following documentation needs to be provided:
- UT documentation
- Build checklist with additional artefacts
- Code Review to be done by one of the development Architects or senior developers.

Onboarding of Development Resources

While onboarding new resources to the development team it is mandatory for them to read, understand and acknowledge the Development Approach document and relevant Standards.

This acknowledgement is a pre-requisite to approving the SAP user provisioning.

Development Standards and Guidelines

Every development will utilize at least one of the supported technologies and their tool sets. Details about the available tools, how to use them at Syensqo and which standards to follow are explained in separated guideline and standard documents.

Standard and Guideline	Link
SAP Development Standards	SAP Development Standards
SAP Integration Development Standards	SAP Integration Development Standards

Workflow history

This view shows the 5 most recent entries. The complete workflow log is available from the 'Document Activity' menu item.

Mar 18, 2026	Actor	Type	Activity	Version
Approved	WENNINGER-ext, Sascha	State	changed state to Approved at 8:43 am	v33
Edited following Approval	WENNINGER-ext, Sascha	State	gave *Minor change* approval at 8:43 am *Additions to BTP content*	
From Feb 11, 2026 to Mar 06, 2026				
	WEINERT-ext, Patrick	Edit	updated the page at 7:21 am	
	WEINERT-ext, Patrick	State	changed state to Edited following Approval at 6:21 am	v30
Nov 12, 2025				
Approved	CHIEW-ext, Yock Sang	State	changed state to Approved at 12:01 pm	v29