SAP Integration Development Standards

Introduction

This guide provides high-level development standards and best practices for SAP Integration Suite modules - SAP Cloud Integration (CI, a.k.a Cloud Platform Integration - SCPI) and API Management (APIM) projects. It is intended for integration developers working on complex integrations between SAP and non-SAP systems. Following these guidelines will ensure consistency, reliability, and maintainability across integration flows.

The guiding principle for programming standards and guidelines at Syensqo is to use what is generally accepted by the industry as “best-practice”, rather than defining a bespoke set of rules. By adopting this approach, it is more likely that developers engaged by Syensqo are already familiar with the “best-practice” approach and can work effectively in the Syensqo environment immediately.

This document’s purpose is to provide developers with the standards and guidance required to develop in Syensqo’s landscape.

> **FIGAF Implementation**
>
> The SyWay Program is actively assessing the adoption of the FIGAF Integration Automation Tool to enhance the management of SAP Cloud Integration (CPI) artefacts, including version control, transport automation, regression testing, and change tracking. As a result, certain guidelines and procedures defined in this document—particularly those governing artifact deployment, testing, and transport sequencing—are subject to change pending FIGAF implementation. Future revisions to this standard will incorporate FIGAF-aligned practices where applicable to ensure consistency and automation across environments.

Assumptions

- All tools required to develop best-practice based are available.
- Since SAP development tools and approaches are evolving so does this document.
- The SAP Development Approach has been understood.
- Integration Suite is a technical platform that generally business users do not access, hence the content is targeted towards the Technical /Developer Community within Syensqo.

Architectural Principles - SAP Best Practice vs Other Integration Platforms

SAP best practice does not always align with "traditional" API-led approaches or other integration architectural principles. The following is an account of a differences of note with reference to the current documentation in Syensqo.

Cloud-native (Non-SAP) Integration

In most **nonSAP, cloudnative integration playbooks**, the guiding assumption is that every boundedcontext or microservice owns both its code **and** it s data store. Platforms supply lowlevel building blocks—managed API gateways, message queues, event buses, statemachine orchestrators, and dataflow services—that architects combine freely to meet performance, latency, or scaling goals. Patterns lean heavily on asynchronous, eventdriven communication so that services can evolve and deploy in isolation. Guardrails for security, compliance, and operations are automated through infrastructureascode, finegrained identity policies, continuousdelivery pipelines, and federated observability that pushes metrics, logs, and traces into a shared telemetry backbone; crossteam consistency is achieved more by contract tests and versioned APIs than by an enterprisewide canonical model.

NonSAP, cloudnative integration guides normally organise recommendations around three layers:

1. **IntegrationDomains** – e.g., applicationtoapplication (A2A) inside a single enterprise, B2B partner connectivity, data & analytics pipelines, and edge/IoT.
2. **IntegrationStyles** – APIled (request/response), event streaming (publish/subscribe), bulk or batch data movement, and workflowbased orchestration.
3. **Usecase Patterns** – *APImanaged* exposure through a gateway that enforces policies and versioning; *eventdriven* messaging that lets looselycoupled microservices react in real time; and *processautomation* workflows or state machines that stitch multiple services into an endtoend business scenario. The guidance encourages each bounded context to own its data, automate guardrails through IaC and CI/CD, and rely on contract tests—not a global canonical model—for semantic alignment.

SAP Centric Cloud Integration

SAPCloud Integration (part of SAPIntegrationSuite) starts from the opposite pole: the SAP digital core (e.g., S/4HANA, SuccessFactors, Ariba) remains the single source of truth, so integrations revolve around prebuilt, upgradesafe artefacts called *iFlows* , a businessaware event mesh, and canonical object definitions (OneDomainModel). “Microservices” are usually sidebyside extensions on SAPBTP or Kubernetes that invoke the core via versioned OData APIs and react to SAP business events without duplicating master data. The Integration SolutionAdvisoryMethodology (ISAM), IntegrationAdvisor mappings, and principalpropagating security guardrails keep interfaces semantically consistent, auditready, and upgradeproof. In practice, organisations often pair SAP’s approach—optimised for enterpriseprocess integrity—with the cloudnative model—optimised for highvelocity, domainowned services—bridging the two via generic connectors or event pipes when a use case spans both worlds.

SAPCloudIntegration (part of SAPIntegrationSuite) applies the same three lenses but prioritises the SAP digital core as the system of record.

- **IntegrationDomains** ISAM defines canonical domains such as CloudtoOnPrem, B2B/EDI, and MasterData sharing.
- **IntegrationStyles** - Recommended styles include APIfirst exposure via versioned OData services, eventmesh–based choreography around SAP business events, batch data replication through prebuilt adapters, and BPMNstyle orchestration inside iFlows.
- **Usecase Patterns** - Corresponding patterns are:
    - *APImanaged* (APIManagement with principal propagation),
    - *Eventdriven* (SAPEventMesh broadcasting CloudEvents that extensions consume without copying master data), and
    - *Processautomation* (graphical iFlows or lowcode WorkflowService). Governance artefacts—naming conventions, version gates, OneDomainModel payloads—ensure semantic consistency and upgrade safety. In hybrid landscapes, organisations often pair SAP’ s domaincentric, upgradesafe model with the highvelocity, domainowned patterns from cloudnative guides, bridging the two through generic connectors or event pipes where processes span both worlds.

Syensqo's IT Integration Team has a detailed Guide and includes an IT Integration Design Framework - see IT Integration Design Framework, extending the capabilities based on SAP's Integration Solution Advisory Methodology (ISAM).

SAP Cloud Integration

Design Principles and Modularization

RULE	**As a Guiding Principle, follow Syensqo's Integration Design Process to identify and use Reference and Use-Case Patterns, with SAP's Integration Solution Advisory Methodology.**

Design integration flows with simplicity, modularity, and maintainability in mind. Key principles include:

- **Integration Packages:** Always group artefacts to Packages based on the Business Process and participating Systems. While it is important to group similar artefacts - e.g. Employee Data Replication - in one package, create multiple packages through other parameters if the number of artefacts is too high.
- **Modularize Flows:** Avoid monolithic, overly complex iFlows. Break down complex processes into smaller, logical units. Use **Local Integration Processes** (sub-processes within an iFlow) to encapsulate reusable or distinct logic blocks. This makes the main integration process easier to read and maintain. If an iFlow is becoming lengthy or handling many tasks, it’s a sign to split it up.
- **One Interface, One IFlow:** Design each iFlow to handle a single integration interface or a specific sender-receiver pair. If an integration scenario involves multiple target systems, consider using **one iFlow per receiver** for clarity and fault isolation. Similarly, if multiple sources send similar messages, you can introduce a **dispatcher iFlow** that routes incoming messages to separate iFlows for each target/system. This separation improves transparency and monitoring since each iFlow represents a distinct interface. (Decoupling flows via asynchronous queues like JMS can further isolate failures and facilitate retries between parts of a process.)
- **Reusable Subflows:** For common sequences (e.g., data enrichment, calling a common API, or error handling routines), consider creating reusable integration processes or even separate template iFlows that can be invoked via the Process Direct adapter. This avoids duplicating logic across iFlows and centralizes updates to that logic.
- **Reusable Artefacts**: Artefacts - e.g. Scripts, Data Types, etc - that are used by multiple Integration Flows in a package should be created as Integration Artefacts rather than uploading within each Integration Flow.
- **Layout and Readability:** Maintain a clean and logical layout in the iFlow editor. Arrange processing steps left-to-right and top-to-bottom in sequence. Use straight connectors and alignment tools (Auto-Layout) to produce a tidy diagram. This visual clarity helps any developer quickly understand the flow. Additionally, make use of labels and annotations: for example, label branches in a router with the condition name, or add notes for complex logic. Clear design and layout will make troubleshooting and future enhancements much easier.
- **Externalize Configurations:** Design iFlows to be environment-agnostic. **Do not hardcode** environment-specific details (URLs, credentials, file paths, etc.) inside flows. Instead, externalize these parameters so they can be configured per environment (Dev/QA/Prod) without altering the flow’s logic. For example, define the endpoint URL or API keys as externally configurable parameters. This makes deployments to higher landscapes simpler and less error-prone and adheres to the principle of 12-factor app configuration.

By adhering to these design principles, you enhance the scalability and maintainability of integrations. The goal is to build flows that are easy to understand, modify, and extend, following solid software design practices adapted to integration scenarios.

Avoid	Timers - Integration triggering the data flow
Avoid	Data cache, transient data only can be subject of data cache for retry purposes only

Avoid	Developing overly complex integrations with multiple sub-processes

Mapping and Transformation

Integrations often require transforming data between formats (XML, JSON, CSV, etc.) or structures. Here are best practices for message mappings and transformations:

- **Choose the Right Tool for the Job:** Use out-of-the-box transformation tools whenever possible instead of custom code, while considering the complexity and supportability.
    - **Graphical Message Mapping**: Use for simple transformation of defined message types, e.g. one-to-one, simple data conversion from XML to JSON, date format changes. A simple transformation will not have complex context handling, and should be easily understood. If many User Defined Functions (UDFs) are required, that is also a good indication that it is not a simple transformation.
    - **XSLT Mapping**: For structured XML-to-XML mapping with complex transformation requirements, leverage **XSLT** mapping – these are optimized for XML and provide a visual mapping interface.
    - **Groovy scripts:** Use for instances that cannot be handled with either Graphical Mapping or XSLT, e.g. where complex computations or dynamic logic needs to be applied, creating dynamic data structures. This approach ensures maintainability - mappings are easier for others to understand and adjust than large script code.
    - **Do not use JavaScript**: Groovy is faster and easier to code than JavaScript, and Integration Developers are overwhelmingly using Groovy.
- **Keep Mappings Manageable:** In graphical mappings, maintain a clean structure. Map only the required fields – use the Filter or Remove contexts functions to drop any unnecessary data early. Keep an eye on mapping complexity: if a single mapping becomes too complicated (e.g., with many functions or if/else logic), evaluate if it should be broken into multiple mapping steps (such as a two-step mapping) or complemented by a script for the complex portion. Simplicity improves performance and clarity.
- **Use Value Mappings and Lookups:** Often you'll need to map code values (e.g., country codes, status codes) between systems. Use the **Value Mapping** artifact for this purpose, which acts as a lookup table for cross-reference values. Populate value-mapping tables with source-to-target value pairs rather than encoding such logic in the mapping script. This separates configuration from logic, and by maintaining these artifacts centrally makes updates easier when values change and enable re-use by multiple IFLows. At runtime, the mapping step can call these by the valueMapping function. Note. Value Maps should be used for technical purposes only if the underlying data is volatile in nature "callbacks" should be considered.
- **Test Transformations Thoroughly:** Develop mappings with sample payloads from real systems. Use the built-in **Mapping Simulator** (for graphical mappings) or external tools for XSLT to validate that your transformations work as expected (especially for edge cases or optional fields). This will catch issues early. Ensure to handle exceptions in mapping – for instance, if an unexpected value appears, you might map it to a default or throw a controlled error that can be caught by error handling logic.
- **Avoid Unnecessary Complexity in Scripts:** When using Groovy for transformation, keep the script focused and as simple as possible. Do not replicate features available in mapping steps or adapters via scripting. For example, **do not use a script to split messages** or to perform simple field mappings that the mapping step can handle. Scripts should ideally be small utilities (e.g., custom date conversion, complex string parsing) within an integration flow. Overusing scripts can make maintenance harder and can introduce performance overhead if not carefully written.

By following these guidelines, you ensure that data mappings are efficient, transparent, and easy to maintain. The key is to use BTP Integration Suite's rich palette of transformation tools to their strengths and keep custom code to a minimum.

RULE	**For Scripting, use only Groovy - do not use JavaScript**

Error Handling and Retry Mechanisms

Robust error handling is essential in integration scenarios to ensure issues are caught and addressed without data loss. SCPI does not automatically retry or store failed messages, so developers must build error-handling into their iFlows:

- **Exception Subprocess:** Take advantage of the **Exception Subprocess** in integration flows. This subprocess (marked with a red border in the design) triggers when any unhandled exception occurs in the main flow. Inside the exception subprocess, implement steps to process the error – for example, log the error details, send an alert notification (via email or HTTP call to an alerting system), or route the failed message to a holding queue/data store. Ensure every critical iFlow has an exception subprocess to catch errors; without it, failed messages might simply get stuck in error status without any automatic notification. A basic pattern is: Exception Subprocess -> Gather error details (like error message, payload snippet, etc.) -> send to an alert channel (email, Slack, etc.) or support ticket system. In some cases, you might also generate an error response back to the sender if it’s a synchronous scenario.
- **Error Logging and Monitoring:** Within error-handling, capture enough information to debug later. For instance, log a unique identifier of the message (like the message ID or a business ID) along with the error description. However, be cautious not to expose sensitive data in error logs. Use general descriptions or masked values for any confidential fields (see Security considerations below on data masking).
- **Retry Strategy:** Determine if and how failed messages should be retried. For transient failures (like a temporary network glitch or target system downtime), a retry mechanism can greatly improve reliability. SCPI doesn’t retry on its own, but you can design a retry logic. Two common approaches:
    - *Sender-based Retry:* In synchronous scenarios, if an iFlow returns an error back to the source, the source system (if capable) should handle re-send attempts. This is often the case with APIs where the caller can retry on HTTP 500 errors. Ensure your documentation tells the source what to do on failure.
    - *Integration-Driven Retry:* For asynchronous flows, or where the sender cannot retry, build a retry loop. For example, use a **Data Store** to persist failed messages, and create a separate “retry iFlow” (perhaps scheduled or triggered via a control message) that picks up these stored entries and attempts to reprocess them. Alternatively, utilize a **JMS queue** for transient errors: send the failed message to a JMS queue with a redelivery policy. The message can be retried from the queue a certain number of times automatically. If you implement this, ensure idempotency – the processing should handle duplicate retries gracefully.

    - Decide on a maximum retry count and interval (e.g., try 3 times with 5 minutes gap) to avoid infinite loops. If after final attempts it still fails, move the message to a dead-letter storage and alert support.
- **Central vs Local Handling:** For organizations with many iFlows, you might create a **global error handling template** – for example, a common exception subprocess design or even a separate error-handling flow that multiple iFlows call (using a Process Direct or JMS to pass the error). This can standardize how errors are processed (uniform email format, centralized error log, etc.). However, ensure this doesn't become a single point of failure. Each iFlow's error subprocess could call the central handler flow, for instance.
- **Testing Error Scenarios:** As part of development standards, include testing of error paths. Simulate errors (e.g., by calling a wrong endpoint or forcing an exception in a script) to verify that your exception subprocess catches them and that the alert/notification mechanism works. This is crucial for critical integrations, so that when a real error happens in production, you are confident it will be caught and reported.

In summary, be proactive in error handling: design your iFlows to fail gracefully. A failed message should not disappear silently; it should trigger a controlled sequence that either automatically fixes the issue (retry) or alerts someone to take action. Good error handling and retry logic greatly increase the robustness of your integration landscape.

ALWAYS	**Handle Errors Gracefully, approach should be driven by NFR's and Interface priority**

Security Considerations

Security is paramount in integration development. SCPI interfaces often handle sensitive data and connect to internal and external systems, so developers must build security into every layer of integration design:

- **Authentication Best Practices:** Use the most secure authentication methods available for each adapter/connection. Preference is to use **OAuth tokens over basic user/password credentials** for connecting to third-party systems. For example, if connecting to an API, use OAuth 2.0 flows or mutual TLS if supported, instead of sending basic auth. Manage certificates and keys in the **Secure Keystore** of the Integration Suite. Regularly update and rotate credentials. Do not hardcode any secrets in Groovy scripts or content modifiers. Instead, store credentials in **Secure Parameters** or the **Credential store** and reference them by name. Certificate-based authentication (client certificates) can be used but due to management overheads it is seen as sub-optimal and should be avoided.
- **Data Encryption:** Ensure data is encrypted in transit. Always use secure protocols (HTTPS for web services, SFTP for file transfers, etc.). SCPI will handle the TLS encryption for adapters like HTTP, but be mindful to check certificate validity and host verification settings. For data at rest (like when using data stores or JMS queues), remember that sensitive data could persist temporarily; design with the assumption that those stores should be treated as secure storage – do not leave unencrypted confidential data indefinitely in them. If you need to persist sensitive data, consider encrypting the content (SCPI supports PGP encryption/decryption steps if needed).
- **Data Masking and Logging**: As mentioned in Logging, never expose sensitive information in logs or error messages. This includes personal data (names, addresses), identifiers (social security numbers), or confidential fields (passwords, API keys). Use masking techniques – for example, log only the last 4 digits of an ID, or replace characters with "*" except a prefix. SCPI's logging does not automatically mask data, so it's the developer's responsibility to ensure that any logged content or exception message sanitizes sensitive details. Also, if you generate files or emails in error handling, scrub them of secrets (e.g., do not email out a full failed payload if it contains sensitive data).
- **Principle of Least Privilege:** When connecting to other systems (cloud or on-premise), use accounts or credentials with minimal necessary permissions. If an iFlow only needs to perform a specific API call, the technical user for that API should not have broader access than needed. Likewise, within SCPI tenant roles, ensure developers and administrators have appropriate roles – e.g., only authorized people can deploy or view messages. While this is more of an administrative concern, developers should be aware and not, for instance, program their iFlow to send data to an unvetted external endpoint.
- **Secure Development Practices:** Validate all inputs even in integration flows. For example, if your iFlow accepts data from an external partner, consider using the Validator step or script checks to ensure the payload is expected (to avoid processing malicious content). Use adapter built-in checks (like schema validation for XML) where applicable. Additionally, be mindful of content-based attacks (e.g., XML External Entity attacks); use the XML validator or parser options that mitigate known vulnerabilities.
- **Utilize SAP Security Features:** SCPI provides features like *PGP encryption, XML Signature, secure parameterization,* and integration with SAP's *API Management* for policy enforcement. Where appropriate, use these. For instance, if you have an open API on SCPI, front it with API Management to apply rate limiting or IP filtering policies. Use the *OAuth2 Authorization Server* in SCPI for client authentication when exposing HTTP endpoints to partners, instead of custom header tokens. Also, if data is highly sensitive, evaluate using the SAP Data Masking or field encryption solutions in the source/target systems so that even the integration layer sees only encrypted values that it passes through.
- **Credentials should not be shared between different systems:** External system should have individual credentials and unique roles assigned to their associated interface, allowing for individual control and management and reducing the risk of "cross talk". Whist many systems in the Syensqo landscape would be regarded as trust worthy, their credentials should be limited to functionality required not generalised access to all endpoints.

By following these security standards, you ensure that integrations do not become the weak link in your enterprise security. In summary: use strong authentication, encrypt data in motion (and at rest if needed), do not expose sensitive info, and code defensively. Always stay updated on SAP's security advisories for SCPI and apply patches or updates to adapters and libraries promptly. Security should be an integral part of your development process, not an afterthought.

ALWAYS	**Apply the most secure Authorisation mechanisms to all interfaces, never assume another system/user is trustworthy**

Web based tools and LLMs

Public web based tools can pose significant security risks, and should not be used for code generation, test data generation, data formatting or testing in general. Any tools should be Syensqo sanctioned applications that are delivered with the application portfolio or trusted software locally installed.

Any use of LLM based coding agents must be avoided. Syensqo's AI Policy defines guidelines on the use of AI tools and strictly prohibits the use of non-approved AI tools.

RULE	**Avoid Web based public tools for code generation, data manipulation and testing**

Documentation

Versioning and Documentation Practices

Maintaining clear version history and documentation for your integration flows is crucial for team collaboration and lifecycle management:

- **Versioning of IFlows:** SCPI allows you to Save versions of an integration flow. Adopt a practice of saving a new version at every significant change or release. After development and testing of a change is complete, **save the iFlow as a version and include a descriptive version comment**. The comment should summarize what changed (e.g., "v1.1 – Added error subprocess for handling timeouts" or "v2.0 – Updated mapping for new field X"). This built-in version history helps in tracking changes and rolling back if necessary. It's also beneficial when multiple developers are involved; everyone can see what the latest changes were. If your team uses an external source control or transport mechanism, ensure the version in CPI aligns with that external tracking.
- **Documentation:** Every integration artifact should be accompanied by clear documentation, either within the tool and in external documents (or both). Leverage the **Description fields** in integration flows and packages: the package description can outline the purpose of the group of iFlows, and each iFlow's description can detail the interface's scope (e.g., "This iFlow sends newly created Orders from SAP S/4 to Salesforce in real-time"). Provide key information like source/target systems, data format, frequency, and any special logic in that description. This is very helpful for onboarding new developers. For complex scenarios, you can attach documents in the package (SCPI allows adding text or attachments) or provide a link to an internal wiki/SharePoint where detailed design docs resides. For instance, a flow that does a complex mapping might have an attached mapping specification document.
- **Maintain an Integration Catalog:** As projects grow, maintain a high-level catalog/LeanIX of all integration flows with their versions, owners, and a brief description. This acts as an index for anyone needing to find or update an interface. It's also useful for impact analysis (e.g., "which interfaces are affected if we change system X?").
- **Coding Standards Documentation:** In addition to functional documentation, document any coding standards or guidelines (some of which are covered by this guide) in a central place. For example, have a checklist that every iFlow must have: proper naming, exception subprocess, no hardcoded credentials, etc. This serves as a QA reference before deploying flows. A peer review process can be instituted where one developer reviews another's iFlow against the checklist.
- **Comments in Artifacts:** While SCPI's graphical interface doesn't have code comments like traditional code, you can insert Annotations (note elements) on the canvas to explain a section of the flow. Use these for any non-obvious logic. In scripts, **comment your code** generously – explain any algorithm or workaround in the Groovy/JS so that someone reading it later understands the intent.
- **Design Guideline Execution:** For Integration Flows, make sure that the Design Guideline tab is executed and **applicable Guidelines are compliant**. Where compliance to an applicable Guideline is not possible, the status need to be marked as skipped and reasoning provided in the information section.
- **Attachment of Test Cases:** (FIGAF) It can be helpful to save sample input/output files or test cases for each iFlow (perhaps in an associated documentation or repository). That way, when modifications are made, developers can re-run those sample cases to ensure nothing broke. While not an out-of-the-box feature of SCPI, this practice of keeping example payloads and expected results is part of good documentation.
- **Specifications** should always be maintained as deep links at **package level** as well as maintaining **Jira card ID's** at individual **iFlow level.**

By rigorously versioning and documenting, you create a knowledge base that supports long-term maintenance. Junior developers onboarding to the project can refer to previous versions to understand how an iFlow evolved and read the documentation to grasp the design decisions. This reduces dependency on oral knowledge transfer and prevents loss of information when team members roll off.

ALWAYS	**Document inline with description labels - do not leave default names**
ALWAYS	**Use Save as Version and add Change Log when releasing a significant change**
ALWAYS	**Execute Design Guidelines in IFlows. Add explanations where compliance is not possible**

Logging and Traceability

Implement logging thoughtfully to aid debugging and auditability, without compromising performance or security. Traceability ensures you can follow a message's path and diagnose issues when they arise:

- **Use Message Logging Wisely:** SCPI offers a **Message Log** (accessible in monitoring) and a **Log Message** step (in integration flow). You should log key events or important data points, but avoid logging entire payloads in production flows. Large payload logging can consume resources and possibly expose sensitive data. *Never use data stores or giant Groovy scripts to log full payloads for debugging* – this is an anti-pattern. If you need to record payload content for troubleshooting, consider writing just a portion (e.g., an order ID, or first 100 characters of a message) or use the built-in **Trace** feature during development/testing. The trace mode in SCPI, when activated, captures detailed payloads and step-by-step details, which is useful for debugging; however, **enable trace only temporarily** in non-production environments or for short periods in production if absolutely needed. Remember to turn it off, as leaving trace on will impact performance and potentially write sensitive data to logs.

- **Add Business Identifiers:** To make tracking easier, incorporate business-specific IDs or correlating keys into your logs. A best practice is setting the SAP standard header **SAP_ApplicationID** or a similar property with a business identifier (like an Order Number, Employee ID, etc.) early in the flow. This ID will then appear in the message monitoring view, allowing you to search for messages by a functional key. It greatly aids operations teams in finding the exact message related to an incident. For example, if an order integration fails, having the Order ID in a searchable field means support can quickly locate that message in the CPI monitor.
- **Correlation for Multi-Step Processes:** If your scenario involves multiple iFlows (e.g., one iFlow splits messages and others process the parts), ensure you propagate a correlation ID across them. This could be done by maintaining a common property or using the message ID. SCPI's message monitoring doesn't automatically group related messages, so it's up to the developer to pass along an ID (perhaps in a custom header like X-CorrelationID). That way, logs in different flows can be tied together when analyzing end-to-end processing.
- **Use of External Logging/Monitoring:** For enterprise projects, consider integrating SCPI with a central logging or monitoring solution. SCPI allows access to message logs via OData APIs – some projects export error logs to an external SIEM or monitoring tool. At minimum, ensure that failed message alerts (discussed in Error Handling) are sent out so someone is notified to check the logs.
- **Non-Repudiation and Audit Trails:** If required by your scenario, use the SAP Cloud Integration's logging capabilities to store an audit trail of messages (e.g., using the Write Variable step to store checkpoints or using persistent data store entries). For example, you might log an entry when a message is successfully processed (with timestamp, IDs, etc.) to an audit database. This can be useful for critical transactions. However, balance this with performance – writing to external systems for every message can slow throughput.

In essence, log enough information to trace what happened for each message, but not so much that it floods the system or exposes sensitive data. Strive for clear, concise log entries that will make sense to someone troubleshooting the interface later. Good traceability means any message can be followed from start to finish, and issues can be pinpointed quickly.

RULE	**Do not log entire payloads in CPI - External logging tools may be used for this purpose if in scope**
RULE	**Never Log Sensitive Information in Customer Header Properties**
ALWAYS	**Use logging functions like messageLog.addCustomHeaderProperty for important data points**

Naming Standards

Consistent naming conventions are critical for clarity and team collaboration. All integration artifacts (such as packages, iFlows, mapping artefacts, etc.) and variables should have meaningful, self-explanatory names.

RATIONALE	**A well-defined naming scheme improves readability and acts as built-in documentation, reducing onboarding time and errors. It allows developers to quickly grasp an artefact's purpose and eases maintenance/troubleshooting.**

- **Integration Packages**
  - **Packages from SAP Catalog**: When copying for Implementation, prefix the Package name - e.g Ariba Network with SAP Business Suite, SAP Fieldglass Integration with SAP SuccessFactors Employee Central
  - **Custom Packages**: Prefix with System Names when required - e.g. Vendor Integration, SuccessFactors and S4HANA Employee Data Integration
- **Integration Artefacts**
  - **Integration Flows - from SAP Catalog**
    - **Do Not Modify SAP delivered standard IFlows** - Changed IFlows will not receive updates issues to the Integration by SAP
    - If the IFlow needs to be changed, copy the IFlow to a Custom Integration Package and modify
  - **Custom Integration Flows**
    - InterfaceID, Source and Target Systems and Integration Action, separated by Underscores. Such naming allows team members to immediately understand an iFlow's purpose and involved systems.
    - E.g. Ariba_to_Keevlar_ReplicateOrderRequest.
  - **Value Mappings**:
    - System Identifiers (if known) and a short description.
    - E.g. S4HANA_to_Ariba_CountryCodes.
  - **Other Artefacts:**
    - Clear description.
    - E.g. - API GetCountryCode
- **Internal IFlow Steps:** User action based short but descriptive texts. For instance:
  - *Message Mapping:* **Map Sales Contract, Map Employee to User**
  - *Content Modifier:* **Set SOAP Header Properties**
  - *Groovy Script:* **Transform Sales Contract to Vendor**
  - *Request-Reply:* **Update Service Contract**
- **Variables and Properties:** Name message properties, headers, and variables with clear intent and scope.
  - Use descriptive keys like OrderID, SourceSystem, TimestampUTC.
  - If using environment-specific parameters, use consistent naming (and consider prefixing or suffixing them to indicate their purpose, such as URL_SAP_ECC for an endpoint URL).
  - **Do not use** generic names like "Var1" or "temp" which don't convey meaning.

RULE	**Never use Project Names (e.g. Syway) in Artefacts names, either in abbreviated form or complete names**
RULE	**Avoid conflicts with SAP Standard naming.**

API Management

API Design Principles

Design APIs following RESTful principles for clarity and consistency. Model your API around **resources** (nouns) rather than actions, and utilize standard HTTP methods (GET, POST, PUT, DELETE) for operations. This approach makes APIs intuitive and easy to use. Resources should be logically modelled (e.g. “customers”, “orders”), with hierarchical nesting only when it reflects real relationships. Always include an API **versioning** strategy from the start: for example, prefix URLs with a version (e.g. /v1/) so that breaking changes can be introduced in a new version without disrupting existing clients. Ensure that new versions of an API can run in parallel with old versions during transitions, giving consumers time to migrate (see for example). Keep APIs **stateless** – each request should contain all information needed, making the service easier to scale and maintain.

Security Standards

Security is paramount in API development. All APIs should enforce strong authentication and protect sensitive data.

- **Authentication & Authorization:** Leverage API Management’s built-in support for API keys and OAuth 2.0. **API keys** can be issued via the developer portal and verified on each call to ensure only registered apps access your API. **OAuth 2.0** provides token-based security for user or service authentication; SAP API Management can validate OAuth 2.0 tokens (e.g. bearer tokens) via its policies. Whenever possible, use OAuth2 for enhanced security (e.g. user-specific access scopes) and API keys for server-to-server or trial access. If needed, implement **Basic Authentication** for simple internal use-cases or legacy support, but prefer more secure methods for public APIs.
- **Client Certificates (TLS):** Only use TLS or mutual TLS (mTLS) where necessary. SAP API Management allows you to authenticate clients using X.509 client certificates. In this setup, clients and servers (mTLS) or the server (TLS) must present a trusted certificate to connect.
- **Data Protection:** Never expose sensitive information unnecessarily. Mask or encrypt sensitive data in transit and at rest. All external API traffic must be over HTTPS to encrypt data over the wire. Avoid logging confidential details (like personal data or credentials) in plain text. If such data must be logged or included in responses, apply masking policies to redact values. For example, you might mask credit card numbers or redact personal identifiers in debug logs and responses. Ensuring GDPR and privacy compliance by design – *only return necessary data fields and anonymize or truncate where appropriate*.

Policy Usage (Traffic & Security Policies)

SAP API Management policies provide powerful ways to control and modify API behavior. Apply policies prudently to enforce standards and protect your services:

- **Rate Limiting & Quotas:** Use traffic management policies to prevent abuse and ensure fair usage. For example, apply a **rate limit or quota** to throttle requests – e.g. 1000 calls per minute or as defined by the API plan. This protects backend services from overload and ensures one client cannot monopolize the API. Quotas can be tied to commercial plans (free tier vs. paid tier) to enforce contractually allowed usage. Generally in context API's are not commodified in a meaningful way, in this context the intent is to stay within limits defined by the underlying service.
- **IP Whitelisting / Access Control:** Implement IP filtering policies to restrict access to trusted networks or clients. By **whitelisting** approved IP ranges (or blacklisting known malicious IPs), you add an extra security layer at the gateway. This ensures only calls from allowed sources reach your APIs. For example, an internal API might only accept calls from your company’s VPN IP range. Maintain these lists as your user base changes (e.g. when partners onboard or network ranges update). * Whitelisting should be a "last resort" security feature.
- **Threat Protection:** Enable content-level threat protection policies to guard against malicious inputs. **JSON and XML Threat Protection** policies should be used to enforce limits on payload size, depth, and structure. For instance, you can limit the maximum JSON nesting depth or array length to prevent resource exhaustion attacks. These policies will block or sanitize unexpectedly large or malformed payloads, thwarting attacks like JSON/XML bombs. Additionally, use **Regular Expression Protection** to detect and block patterns of harmful content (such as SQL injection attempts in inputs).
- **Other Policies:** Consider other built-in policies to improve API performance and security. For example, use a **caching** policy to cache frequent responses and reduce load on backends (if the data can be safely cached). Employ **message transformation** policies (XML<->JSON conversions, mapping) to integrate with different backend formats as needed – though keep transformation minimal to avoid high latency. Use the **Assign Message** or scripting policies to inject standard headers (e.g. correlation IDs, CORS headers) across all responses for consistency. Keep policy flows as simple as possible; use only those needed for your use-case to maintain performance.
- **Plans (Rate Plans/Usage Plans): No requirement at this stage in this project.**

Decisions must be made at an individual API level as to appropriate level of security through careful review of requirements, sending system capability and risk profile of the data and integration. This list is meant as a guide of possible security components that **may** be leveraged.

Naming Conventions

Adopt consistent naming conventions for all API Management entities to make them immediately understandable:

- **API Proxies:** Use short, descriptive names that reflect the API's purpose and (if applicable) its version. For example, an API proxy providing employee data version 1 could be named **`Employees-v1`** (if its base path is `/v1/employees`). Avoid spaces or special characters.
- **API Products:** Name products by grouping or domain. A product that bundles customer-related APIs might be called **`CustomerAPIs`** or **`Customer-Services`** . The name should clue the consumer into the API domain or business area. Keep it concise and avoid ambiguous terms..
- **Resource URIs:** Use lowercase nouns for endpoint paths, pluralized where appropriate (e.g. `/customers/{customerId}/orders`). **Do not include verbs** in resource paths – the HTTP method defines the action. Separate words with hyphens for readability instead of underscores. Good URI naming makes the API self-descriptive; for instance, `GET /orders/123` is clearly fetching order 123, and `POST /orders` creates a new order. Consistent, human-readable resource names improve the developer experience.

Organisational Components

Level	Purpose in the lifecycle	Typical artefacts shown to the developer
APIProxy (often shortened to "API")	A façade in front of a backend service. Holds policies, base path and versioning.	Paths such as `/v1/customers`, policy bundles, revision history.
Product	Logical bundle of one or more API proxies that will be published to the catalogue. It is the subscription unit.	Product tile in DeveloperHub, monetisation rateplan, default quota.
Application	Registration made by an external (or internal) developer to consume Products. Generates *App Key* and *Secret*, collects analytics and is the traffic throttling unit.	App credentials, list of subscribed products, call statistics.