SAP PRESS

SAP® Cloud Integration Cookbook

Advanced Cloud Integration with SAP Integration Suite

- Find solutions to complex problems, master best practices, and learn advanced Cloud Integration techniques for your SAP system
- Work with Groovy, XSLT, message mapping, adapters, and more
- Create reusable integration artifacts, design loosely coupled interfaces, and manage transports with CI/CD pipelines

Martin Koch
Thorsten Reisinger
Marc Urschick

Rheinwerk Publishing

Contents

Chapter 5
Loosely Coupled Interfaces

In the rapidly evolving landscape of enterprise IT, agility, scalability, and resilience have become pillars of integration architecture. Traditional integration models, which often rely on tightly coupled systems and synchronous communication, struggle to meet the demands of modern digital ecosystems. When systems are directly dependent on one another's availability, performance, and interface consistency, the risk of bottlenecks, single points of failure, and cascading outages increases significantly. In response to these limitations, organizations are increasingly turning to asynchronous messaging patterns and loosely coupled designs to build robust, future-proof integration solutions.

This chapter explores the principles and practices of implementing loosely coupled interfaces by using *Java Message Service (JMS)* within SAP Business Technology Platform (SAP BTP)—specifically through SAP Integration Suite. Messaging via JMS is a cornerstone of decoupled integration design because it offers an architecture where producers and consumers of data operate independently and are temporally separated. This architectural paradigm not only boosts system reliability and scalability but also aligns closely with event-driven architectures and microservices principles.

SAP Integration Suite, which is a core component of SAP BTP, provides native support for JMS-based messaging. It enables the use of queues and topics to manage communication among systems that may not always be available at the same time or operate at the same throughput levels. By buffering messages in durable queues and handling them asynchronously, integration developers can design flows that are resilient to downstream failures, network fluctuations, and variable workloads.

The intention of this chapter is to equip you with a practical and conceptual understanding of how to leverage JMS for loose coupling within Cloud Integration scenarios. Whether you are designing interfaces between SAP S/4HANA and SAP SuccessFactors, integrating with third-party cloud services, or creating custom APIs that require fault tolerance and queue-based buffering, the patterns we discuss here are applicable across a wide range of use cases.

Section 5.1 introduces the fundamental concepts of messaging and loose coupling. By decoupling the sender and receiver, messaging systems allow applications to evolve independently, communicate asynchronously, and maintain functionality even under

partial failures. This chapter highlights the benefits and trade-offs of this integration style.

Section 5.2 presents the structure of a messaging architecture by describing the essential elements such as message producers, consumers, queues, and topics. It also discusses the roles of brokers and message persistence to clarify how complex systems exchange information reliably without direct method calls.

Section 5.3 turns to the configuration of the JMS adapter, which serves as the bridge between applications and the messaging infrastructure. Proper configuration enables seamless delivery of messages and sets the foundation for secure, performant communication across different technical environments.

Finally, Section 5.4 examines error handling and retry strategies, which are crucial aspects of building fault-tolerant systems. From dead letter queues (DLQs) to backoff policies, this section demonstrates how to design messaging solutions that not only exchange data but also withstand operational challenges gracefully.

5.1 Introduction to Messaging and Loose Coupling

In modern enterprise architectures, integration scenarios often involve multiple systems with different technologies, lifecycles, and availability constraints. Traditional point-to-point or synchronous communication patterns, such as direct HTTP and SOAP calls, tightly couple the sender and receiver systems, and this approach can lead to system bottlenecks, cascading failures, and scalability issues.

Loosely coupled interfaces address these challenges by decoupling the communication between systems. Instead of via direct communication, messages are sent through an intermediary—typically, a message broker or queue—that allows systems to operate independently. In this model, the sender system is only responsible for placing a message in the queue while the receiver processes messages from the queue at its own pace.

SAP BTP offers robust capabilities to support loose coupling through SAP Integration Suite—particularly via the use of JMS, which enables *asynchronous messaging*, in which systems exchange information without direct dependencies. This not only increases fault tolerance and scalability but also improves maintainability and flexibility of integration landscapes.

To help you better understand these concepts, this section explores several of their foundational aspects. We begin with asynchronous processing and explain how decoupled communication patterns help systems achieve resilience and scalability. Next, we introduce the JMS as the standard API that enables such messaging within SAP BTP. Building on this, we outline the key benefits of applying JMS in integration scenarios, highlighting its role in reliability and flexibility. At the same time, it's

important to recognize JMS' limitations, so we also discuss when not to use JMS to help you pick the right tools for the right scenario. The section concludes with best practices for designing loosely coupled interfaces to provide you with practical guidance for building iFlows that are robust, efficient, and maintainable.

5.1.1 Asynchronous Processing

At the heart of loosely coupled interfaces lies the principle of *asynchronous processing*—a communication model that breaks the rigid dependencies between systems by decoupling the sender and receiver in both time and state. Unlike in synchronous interfaces, where the sender waits for a response before proceeding, asynchronous messaging allows each system to operate independently. This approach introduces greater flexibility, fault tolerance, and scalability into enterprise integration landscapes, and it's a foundational concept for modern, resilient architectures.

In a traditional synchronous scenario—such as an HTTP request or a remote function call—the sender and receiver are tightly bound. If the receiving system is slow, unavailable, or experiencing errors, the sender is immediately impacted. This not only affects performance but also propagates instability across the system. Conversely, asynchronous processing introduces a buffer, often in the form of a message queue or event stream, that stores the message temporarily until the receiving system is ready to consume it. This architecture allows for much looser coupling between services, thus improving system uptime and enabling nonblocking communication flows.

JMS plays a crucial role in enabling asynchronous communication within Cloud Integration. JMS queues act as durable message brokers that allow iFlows to send and receive data independently of each other. A producer flow can generate and enqueue a message without needing to know whether the consumer is currently online or available. The consumer flow, on the other hand, can process the message later—when resources become available or at a controlled rate that matches the backend system's capacity. This decoupling of message producers and consumers is one of the key enablers of scalability, as it allows system components to scale independently and handle traffic bursts more efficiently.

Another major benefit of asynchronous processing is improved fault tolerance. Temporary outages in downstream systems no longer result in immediate interface failures. Instead, messages are retained in the queue, retried according to configurable strategies, and optionally redirected to a DLQ if they fail beyond defined thresholds. This enables operational teams to recover gracefully from failures without losing data, while also providing visibility into unresolved errors.

Asynchronous processing also allows for parallelism and concurrency, especially on the consumer side. Multiple consumer instances or threads can process queued messages simultaneously to increase throughput without overloading the source system.

This makes asynchronous messaging ideal for high-volume integrations such as order ingestion, data replication, and system-to-system handoffs.

It's important to note that asynchronous processing introduces new design considerations. For example, since the response isn't immediate, developers must use additional correlation mechanisms (e.g., correlation IDs, reference tokens) track end-to-end message flow. They must also consider message ordering (using EOIO, for example), idempotency, and state management, especially when building interfaces with business-critical requirements.

This approach also serves as a stepping-stone toward more advanced patterns like event-driven architecture (EDA), which we cover in detail in a later chapter of this book. While asynchronous messaging relies on queues or channels for point-to-point or broadcast delivery, event-driven systems elevate this concept by using events to signal changes in system state. In an EDA model, systems react to business events—such as a sales order being created or a delivery status changing—rather than being directly told what to do. This enables even looser coupling, greater extensibility, and real-time responsiveness across distributed systems.

In many enterprise landscapes, asynchronous processing via JMS acts as a bridge between traditional, tightly coupled interfaces and more modern, event-based models. It offers a pragmatic and powerful way to incrementally move toward decoupled architectures without overhauling legacy systems all at once. By adopting asynchronous processing patterns within Cloud Integration, organizations can gain immediate benefits in reliability, performance, and maintainability—while also laying the groundwork for more dynamic and intelligent architectures powered by events.

5.1.2 Java Messaging Service

As organizations move toward building more decoupled and resilient integration landscapes, JMS emerges as a key enabler of asynchronous, loosely coupled communication within Cloud Integration. JMS provides a standardized messaging protocol that allows distributed applications to communicate through message queues in a reliable and scalable way—regardless of whether the sender and receiver are available at the same time.

In the context of SAP Integration Suite, JMS acts as a *message broker* that facilitates asynchronous communication between iFlows. The core concept is simple but powerful: a producer flow sends a message to a JMS queue, where it's stored persistently until a consumer flow retrieves and processes it. This mechanism decouples the lifecycle and availability of the producing and consuming systems to ensure that neither side is directly dependent on the other being online or immediately responsive. This is particularly valuable for handling system outages, batch processing, and integrations with systems that operate at different speeds or volumes.

JMS offers the following two delivery modes that cater to different business requirements:

- *Exactly once* (EO) ensures that a message is delivered one time and only one time, which is essential for ensuring data consistency across systems.
- *Exactly once in order* (EOIO) provides both single delivery and strict order guarantees for messages that belong to the same sequence. This is particularly useful for business transactions in which the order of events matters, such as financial postings and shipment updates.

One of the key features of JMS in Cloud Integration is its persistence. Messages stored in a JMS queue are saved in a durable store, which means they are not lost even in the event of system restarts or crashes. This level of reliability makes JMS suitable for mission-critical enterprise processes where data integrity and recovery are non-negotiable.

Beyond their basic message handling, JMS queues offer a range of advanced features that support fault-tolerance and operational flexibility. For example, failed messages can be retried automatically based on configurable retry intervals and attempt counts. If a message still can't be processed after the defined number of retries, it can be redirected to a DLQ for later analysis and manual reprocessing. This built-in error-handling framework ensures that temporary failures don't disrupt the overall message flow and that irrecoverable errors can be isolated without data loss.

Another important advantage of JMS is its ability to handle high volumes and enable parallelism. You can configure consumer flows with multiple concurrent threads to allow the processing of several messages in parallel. This is especially useful in high-throughput scenarios like order processing, master data replication, and IoT data ingestion. Combined with its ability to monitor queue metrics, retry patterns, and message traces, JMS offers both scalability and deep operational insight.

From an architectural perspective, JMS isn't only a messaging engine but also a design abstraction that supports the principles of loose coupling, modularity, and fault isolation. By placing a JMS queue between systems, integration architects introduce a layer that can absorb fluctuations in load, break dependency chains, and improve the overall resiliency of the solution. It becomes possible to evolve systems independently, schedule updates without downtime, and respond more gracefully to failure scenarios.

Moreover, JMS can act as a bridge toward event-based design, especially when used with message enrichment and header-based routing. Although JMS is fundamentally point-to-point, its ability to decouple message producers from consumers makes it conceptually aligned with event-driven patterns, where systems respond to occurrences rather than relying on tightly coupled request/response cycles.

5.1.3 Key Benefits

The adoption of loosely coupled interfaces marks a significant shift in how modern enterprise systems communicate, integrate, and scale. Unlike tightly coupled architectures—in which system components are interdependent and sensitive to changes—loosely coupled interfaces introduce flexibility, independence, and resilience, all of which are vital in the dynamic environments businesses operate in today.

One of the most immediate benefits of loose coupling is *asynchronous communication*, which removes the need for systems to be available at the same time. This temporal decoupling ensures that if a target system is offline, under maintenance, or experiencing load issues, the sending system can still transmit data via queues without interruption. This improves the uptime and stability of the overall landscape, thus allowing each system to operate according to its own availability and performance profile. Systems become more autonomous, and the likelihood of cascading failures caused by a single point of disruption is significantly reduced.

Another core advantage is *scalability*. By decoupling producer and consumer flows with technologies such as JMS, each part of the system can be scaled independently. For instance, during periods of high message volume, additional consumer threads can be added without impacting the sending system. Likewise, message producers can operate at high speeds without overwhelming the receivers, thanks to buffering through queues. This enables organizations to handle seasonal spikes, high-load periods, and bursts of activity without costly overprovisioning or architectural rework.

Resilience and fault tolerance are also greatly improved. Loosely coupled interfaces can gracefully handle transient failures through built-in retry mechanisms, and messages that repeatedly fail can be redirected to DLQs for later inspection. This prevents data loss and ensures that errors don't go unnoticed or silently degrade business processes. With additional support for error categorization and manual reprocessing, teams are better equipped to identify, respond to, and resolve issues quickly.

A key benefit that's often overlooked is *change agility*. In tightly coupled systems, changes to one interface often require coordinated updates across multiple systems—which is a slow, risky, and error-prone process. Loosely coupled interfaces mitigate this risk by isolating changes. A new version of a consumer flow can be introduced independently, or new consumers can be added without impacting the producer at all. This modularity allows teams to innovate faster, deploy incrementally, and respond more quickly to new requirements or regulations.

Moreover, observability and monitoring are enhanced in a decoupled setup. With clear separation of message producers, queues, and consumers, administrators can track message flow through each layer and identify bottlenecks and errors with greater precision. Metrics such as queue size, message age, retry frequency, and throughput provide insights that are useful not only for operational monitoring but also for long-term forecasting and capacity planning.

Another strategic benefit is support for hybrid and multicloud architectures. Loosely coupled messaging systems can span across on-premise and cloud environments to bridge gaps between legacy systems and modern platforms. JMS-based messaging within SAP BTP, for example, allows for reliable and asynchronous communication among SAP S/4HANA, third-party APIs, and cloud-native services, regardless of where they are hosted.

Finally, loosely coupled interfaces lay the groundwork for EDAs. While this book includes a dedicated chapter on EDAs, it's important to note here that the principles of loose coupling, asynchronous flow, and queue-based design naturally extend into event-driven patterns. As businesses move toward real-time, reactive systems that respond to business events rather than procedural triggers, loosely coupled messaging becomes the architectural stepping-stone toward future readiness.

5.1.4 When Not to Use Java Messaging Service

While JMS-based messaging and loosely coupled interfaces offer significant advantages in terms of scalability, resilience, and decoupling, they are not universally suitable for every integration scenario. Like all architectural patterns, their effectiveness depends on the business context, functional requirements, and performance expectations. You must understanding when not to use JMS or loose coupling to avoid overengineering, unnecessary complexity, and unintended side effects.

One of the most common reasons to avoid JMS is because you need real-time, synchronous feedback. If a client system or end user expects an immediate response—such as in login validation, pricing checks, or payment authorizations—a decoupled, asynchronous model introduces unacceptable latency. Synchronous protocols like HTTP or SOAP are better suited to such cases because they support request–response patterns with strict timing guarantees. Implementing JMS for these use cases would complicate the architecture without providing meaningful benefit and could even lead to a degraded user experience.

Additionally, simple, one-off integrations that don't involve complex workflows, long processing chains, or retry mechanisms often don't warrant the use of a JMS queue. For example, you may be able to better implement a basic file transfer between two stable systems or a low-volume API call that rarely fails by using direct point-to-point integration. Adding JMS to these flows could result in increased development effort, monitoring overhead, and operational complexity, without significantly improving reliability or scalability.

Data consistency requirements can also be a constraint. In tightly coupled transactional processes—especially those that require atomicity, consistency, isolation, and durability (ACID)–compliant operations across multiple systems—asynchronous processing introduces a risk of data desynchronization. For instance, in multistep order processing—where each step must succeed or roll back as one atomic unit—synchronous processing

may be necessary to preserve transactional integrity. While JMS provides delivery guarantees like EO and EOIO, it doesn't natively support distributed transactions across systems, which could lead to complex compensation logic or inconsistent states if not carefully managed.

Another consideration is monitoring and troubleshooting complexity. Asynchronous flows with loose coupling make it harder to trace a single transaction end-to-end, especially in environments without mature observability tools or disciplined correlation practices. Teams that are unfamiliar with asynchronous design may struggle to track down errors, which can lead to longer resolution times. In such environments, simpler synchronous flows may be easier to manage and support.

Organizations should also consider skill set and operational maturity. Those that are just beginning their journey with SAP Integration Suite or that don't have experience in managing queues, retries, and dead letter processing might initially benefit from building synchronous flows. Jumping into loosely coupled designs without a solid foundation in message handling and exception strategies can lead to poorly configured retry loops, overflowing DLQs, or message duplication—which can ultimately reduce reliability rather than enhancing it.

Finally, regulatory or security constraints can influence the decision of whether or not to use JMS. In some industries, such as finance and health care, strict audit trails and sequencing requirements may be easier to enforce in synchronous transactions that provide immediate feedback and tightly bound state changes. While JMS can support message logging and traceability, meeting specific compliance mandates may be more straightforward if you use tightly controlled synchronous processes.

5.1.5 Best Practices for Designing Loosely Coupled Interfaces

Based on both conceptual and technical guidance, several best practices have emerged as vital to building effective JMS-based integration solutions:

- Clearly separate producer and consumer flows. Designing producer and consumer flows as independent artifacts allows for cleaner deployments, better maintenance, and flexible scaling. This modularity also simplifies testing and versioning.
- Use EOIO for sequence-sensitive data. Where business transactions—such as financial postings or status updates— depend on strict message order, EOIO ensures that messages are processed in the sequence in which they were sent. This avoids race conditions and state inconsistencies.
- Implement robust error handling. You should incorporate exception subprocesses to manage failures at each critical point in the iFlow, use DLQs as a fallback to capture unresolved errors without halting message processing, and ensure that all failure scenarios are logged and monitored appropriately.
- Optimize resource usage. Be mindful of JMS resource quotas in your SAP BTP tenant. Use shared queues with routing logic when possible to reduce queue count, compress

payloads to minimize memory usage, and offload large files to external storage like SAP Document Management.

- Configure smart retries. You should tune retry intervals and counts based on the expected recovery time of the target system, and you should use exponential backoff to avoid overwhelming systems that are struggling or recovering.
- Implement monitoring and alerting. Proactively track message health, queue utilization, and retry behavior. Set up alerts for thresholds such as queue size, DLQ growth, or retry saturation. Integrate with existing ITSM platforms to ensure that operational teams can act quickly.
- Understand when not to use JMS. JMS is ideal for asynchronous, decoupled use cases. However, for real-time interactions, synchronous request–response needs, and transactionally bound operations, a different integration model—such as direct HTTP or OData—may be more appropriate. Use JMS where it adds the most value.

5.2 Structure of Messaging Architecture

Messaging architecture provides the backbone for reliable, asynchronous communication among distributed systems. By decoupling producers and consumers, it allows applications to scale independently, ensures message durability, and supports fault-tolerant data exchange across services. A well-designed messaging architecture enables flexibility, resilience, and observability in complex enterprise environments.

This section explores the structure of such an architecture in four parts. Section 5.2.1 covers the producer iFlow, which describes how messages are created and published into the system. Section 5.2.2 introduces the JMS queue, which is a central component for storing and routing messages. Section 5.2.3 examines the Consumer iFlow by focusing on how applications receive, process, and acknowledge messages. Finally, Section 5.2.4 covers monitoring and metrics, emphasizing the importance of visibility and performance tracking to ensure reliable message delivery and system health.

As shown in Figure 5.1, the JMS channel is used to connect the source system with the target system. The queue breaks the two sides into separate threads that don't directly connect to each other. This means that the source system can publish its updates uninterruptedly into an inbound heap of events, which are then processed simultaneously while also introducing the possibility of multiplexing the event for multiple target systems at the same time, without duplicating the event.

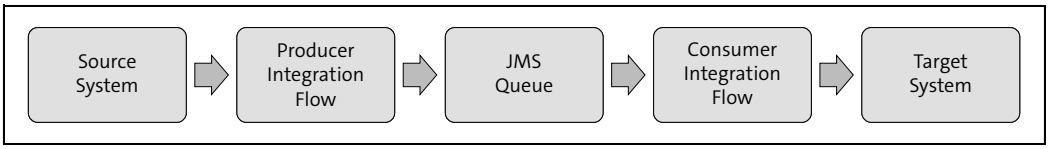

Figure 5.1 End-to-End Messaging Architecture Using JMS Queue between Source and Target System

5.2.1 Producer iFlow

The producer iFlow serves as the starting point of any JMS-based message process in Cloud Integration. Its primary responsibility is to receive data from upstream systems, standardize and enrich the message content, and deliver it into a JMS queue for asynchronous processing. This decouples the sending system from the consuming process to enable better scalability, error resilience, and system independence.

Typically, the producer flow begins by ingesting input from a source system, which could be exposed through a RESTful API, an IDoc interface, a SOAP web service, or an SFTP trigger. This flexibility allows SAP Integration Suite to connect to a wide variety of both SAP and non-SAP applications and services. Once the message is received, it usually undergoes transformation into a standardized internal format so downstream consumers receive data in a consistent and expected structure. You can achieve this transformation by using graphical mapping tools, Groovy scripts, or XSLT mappings, depending on complexity and format requirements.

Next, the message is enriched with essential metadata, often using header properties. These might include a unique message or correlation ID, priority level, business context information, or routing hints—which are attributes that are crucial for traceability, error handling, and message filtering in later stages. Enrichment ensures that each message carries the contextual information that's needed for intelligent routing and robust monitoring.

Finally, the message is handed off to the JMS adapter, which acts as the bridge between the iFlow and the JMS queue. The adapter pushes the message into the defined queue, where it will remain in a durable, persistent state until a consumer flow retrieves it for further processing. By offloading the message into the queue, the producer flow completes its role, freeing up system resources and enabling true asynchronous processing—which is an essential architectural pattern in modern, decoupled enterprise integrations.

This iFlow, as depicted in Figure 5.2, represents an inbound process where a message is received via HTTPS and routed to a JMS queue. The flow begins with a sender system initiating communication through HTTPS. The first step is **Define request**, in which the inbound message is structured. The flow then moves to the **Send to JMS queue** step, which enqueues the message for asynchronous processing. A JMS receiver component subscribes to this queue, and meanwhile, the flow continues by defining a response in the **Define response** step.

It concludes with the **End** event by sending a reply to the sender. This design allows decoupling between the sender and receiver to improve scalability and reliability. The clear separation of steps and the use of JMS ensures reliable message handling in asynchronous integration scenarios using the cloud connector.

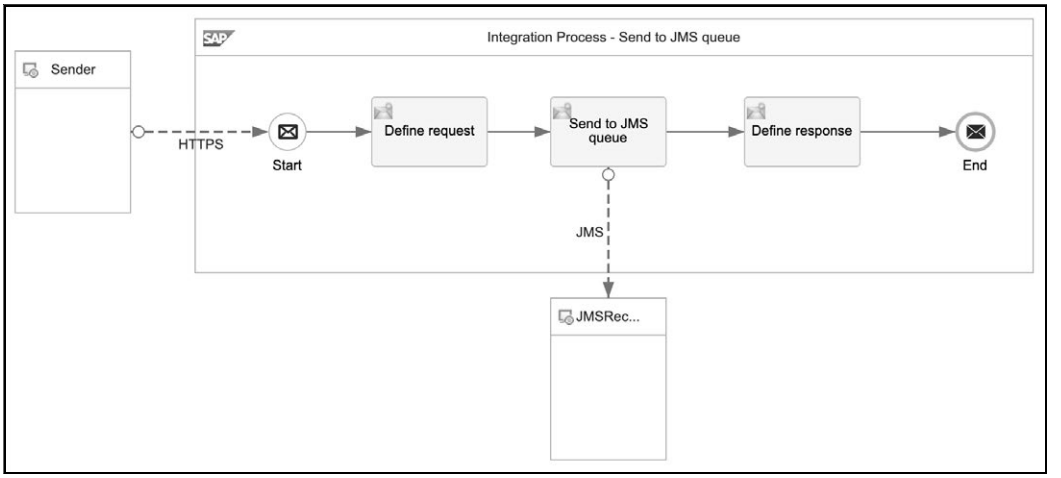

Figure 5.2 Inbound Message Process

Next, you can configure the JMS sender step (see Figure 5.3) by clicking the JMS arrow that points to the JMS Adapter at the bottom. This takes you to the **Processing** tab of the JMS configuration within an iFlow, where you can enter "DecoupleFlows" as the **Queue Name** of the step. Enter "2" in the **Retention Threshold for Alerting (in d)** field to trigger alerts if messages remain for over 2 days. You can set messages to expire after 90 days by entering "90" in the **Expiration Period (in d)** field. Check the **Encrypt Stored Message** box to enable message encryption and ensure secure storage, but leave **Transfer Exchange Properties** unchecked.

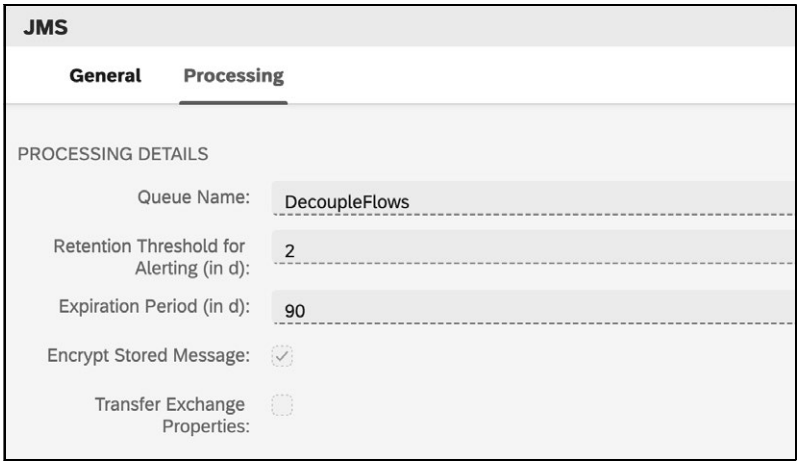

Figure 5.3 Inbound Process: JMS Settings

5.2.2 Java Messaging Service Queue

The JMS queue plays a central role in Cloud Integration's asynchronous messaging architecture. It serves as a reliable buffer between the producer and consumer flows that enables decoupled processing and ensures message durability, even in cases of system downtime or processing delays. Once a message is handed off by the producer iFlow, the JMS queue holds it until a consumer is ready to retrieve and process it. This introduces a critical layer of resilience and load management.

One of the defining characteristics of the JMS queue is its persistent message storage. Messages placed into the queue are stored durably, which means they survive system restarts and remain intact until they are successfully processed or manually removed. This persistence guarantees delivery reliability—which is especially important in enterprise environments where data loss is unacceptable.

JMS queues also support two critical delivery modes: EO and EOIO. EO ensures that each message is delivered a single time, regardless of system failures or retries, while EOIO maintains the sequence of message delivery within a specified group (such as a business transaction or customer ID). These delivery guarantees allow architects to meet strict business and regulatory requirements for consistency and order.

In Cloud Integration, administrators must explicitly configure JMS queues under the JMS resources section of the tenant. There, they define queue names, retention settings, and quotas. It's also important to note that queues are subject to quota and memory limitations, which vary depending on the selected service plan—standard plans allow fewer queues and lower throughput than advanced or enterprise plans. Therefore, you must do careful capacity planning to prevent queue saturation and ensure sustainable operation.

The SAP monitoring dashboard provides real-time visibility into each queue's status, including message count, age, and health indicators. With this tool, integration teams can identify backlogs, monitor throughput, and proactively manage queue behavior, thus reinforcing the critical role JMS queues play in supporting scalable, resilient, and asynchronous integration processes.

5.2.3 Consumer iFlow

The consumer iFlow in a JMS-based architecture is responsible for retrieving messages from the queue and performing the business logic required to deliver them to their final destination. It acts as the second half of the decoupled integration pattern by processing messages independently of the producer flow and offering the flexibility to scale, enrich, and route messages, based on dynamic business needs.

The flow begins with the JMS adapter configured as a sender that continuously listens to a specific queue and pulls incoming messages. As messages are retrieved, the flow typically applies filtering or routing logic, often using JMS header properties or specific

fields within the payload to direct messages along different processing paths. This step is essential when multiple message types or business domains share a queue because it ensures that each message reaches the correct downstream system.

Once routed, messages are passed through transformation components where the data is enriched, validated, or mapped into a format that's required by the target system. This transformation can involve lookups, value conversions, or even complex scripting, depending on the business scenario. Finally, the message is forwarded by the appropriate outbound adapter—such as HTTP, SOAP, or SAP SuccessFactors—to complete its journey to the target application or service.

A defining strength of the consumer flow is its support for parallel processing. By configuring multiple concurrent threads, the system can handle higher message volumes and improve throughput without impacting message order when EOIO isn't required. To ensure robust reliability, the flow typically includes exception subprocesses that handle errors gracefully, thus enabling retries or triggering alert workflows. If a message can't be processed even after multiple attempts, it's automatically routed to a DLQ for post-failure analysis and manual recovery.

In this way, the consumer flow acts as the intelligent processing layer of JMS-based integration. It delivers messages efficiently while providing the resilience, scalability, and fault tolerance that are needed for enterprise-grade scenarios.

The "Poll from JMS queue" iFlow in Figure 5.4 illustrates a JMS-driven processing pipeline within SAP Integration Suite that's optimized for decoupling and asynchronous message handling.

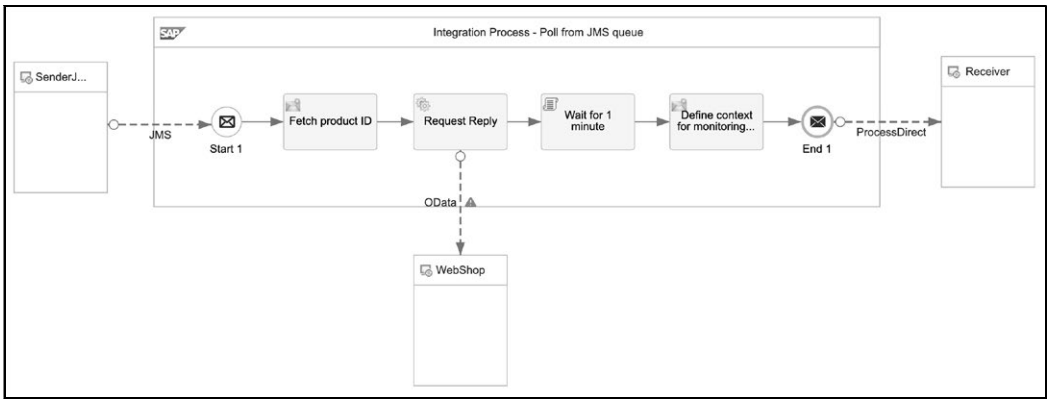

Figure 5.4 Outbound JMS Processing Flow

It begins with a JMS sender adapter that polls messages from a JMS queue, triggering the process at **Start 1**. From there, it proceeds as follows:

1. The **Fetch Product ID** integration step involves extracting or constructing a product ID from the incoming message. This is likely a content modifier or script step that's used to prepare the payload for the downstream service call.

2. Using the product ID derived earlier, the **Request Reply** step sends a request to the WebShop component to retrieve the most recent product data. It follows a request-reply pattern over an OData connection to ensure that the iFlow always processes current and accurate product information.
3. The deliberate **Wait for 1 Minute** simulates a high-compute or time-consuming backend process in real-world scenarios. In practical terms, it represents steps like intensive data transformations, aggregation, and third-party API calls that take substantial time. The actual use might involve a complex computation that can't be shown in a simplified integration design.
4. The **Define context for monitoring** step enriches the message with context properties, which are critical for traceability, alerting, and custom monitoring in runtime tools.
5. The process ends at **End 1**, where the enriched message is handed off via the **Process-Direct** adapter to another iFlow or processing block and is finally sent to the **Receiver** system.

This modular design ensures reliability and traceability in asynchronous, resource-intensive scenarios.

5.2.4 Monitoring and Metrics

Effective monitoring is essential for maintaining the health, performance, and reliability of JMS-based iFlows in Cloud Integration. The **Monitoring** section within SAP Integration Suite provides administrators and support teams with real-time visibility into message flow execution, queue usage, and error conditions. This observability layer plays a critical role in proactively identifying issues, enforcing service-level agreements (SLAs), and ensuring smooth end-to-end operations.

One of the key capabilities is the ability to track messages throughout their lifecycle—from the moment they enter the producer flow, through the JMS queue, and finally to the consumer flow. Administrators can drill into individual message traces to inspect payloads, headers, adapter logs, and retry history. This end-to-end traceability is invaluable when troubleshooting delayed or failed messages because it provides full transparency into where and why processing may have broken down.

The monitoring dashboard also makes it easy to identify stuck or failed messages. Integration support teams can filter messages by status (e.g., **Failed**, **Retried**, **Successful**) and isolate bottlenecks, such as messages that are stuck in retry loops and flows that repeatedly send content to a DLQ. By analyzing patterns in failure rates and exception types, teams can take targeted actions to strengthen specific integration components.

In addition, Cloud Integration enables users to monitor message retries, processing durations, and throughput at both the flow and tenant level. This data is crucial for

detecting abnormal load spikes, degraded performance, and underutilized resources. Advanced users can further correlate these insights with business events (e.g., promotions, financial closings) to forecast future capacity needs.

Finally, the system provides visibility into JMS quota usage and threshold breaches. Alerts can notify operations teams when a queue is nearing capacity, retry limits are being hit frequently, and DLQs are growing unexpectedly. These proactive alerts enable faster response times and help prevent queue exhaustion and message loss.

5.3 Configuration of the Java Messaging Service Adapter

The JMS adapter in Cloud Integration supports both sending and receiving messages to and from queues. It's configured within iFlows to either publish messages to a queue or consume messages from it. Correct configuration is a critical factor in ensuring reliable message delivery, efficient resource usage, and alignment with overall integration requirements.

In the following sections, we'll examine the most important aspects of JMS configuration in the SAP BTP context. Section 5.3.1 looks at the configuration properties for the sender and receiver, which define how messages are published and consumed. Section 5.3.2 highlights the limitations of the JMS queue in SAP BTP to help you make design decisions that account for platform-specific constraints. Section 5.3.3 helps you understand transactional behavior and concurrency so you can build reliable flows that can handle parallel processing without data inconsistencies. Section 5.3.4 discusses queue design strategy, including how to structure queues for scalability and maintainability, and Section 5.3.5 covers the optimization techniques that improve throughput and performance. Finally, Section 5.3.6 addresses common pitfalls that developers should avoid to prevent issues in production scenarios. Together, these sections provide a comprehensive guide to configuring JMS effectively for robust and scalable integration solutions.

5.3.1 Configuration Elements

When using the JMS adapter, choose one of the following access type options to determine how messages are processed, based on your integration needs.

- **Non-Exclusive**
  Choose this access type if you want messages to be processed in parallel across multiple worker nodes, which is suitable for scenarios where message order isn't essential. The JMS adapter won't serialize messages, which means that there will be no guarantee of the order in which the messages will be consumed by Cloud Integration. For additional information, also check out Table 5.1.

- **Exclusive**
  Choose this access type for cases where preserving the order of message processing is crucial. This guarantees that messages will be processed in the exact order they are received by allowing only one consumer to have access to the queue at any time.

Sender Adapter

When configuring how the sending side produces messages into the JMS queue, you can configure some parameters to fine-tune the behavior of the queue. The following list covers how to configure these parameters, which you do on the screen shown in Figure 5.5 (as you'll see when you open an iFlow).

- **Queue Name**
  Enter the name of the message queue.
- **Access Type**
  Select one of the following options:
  - **Non-Exclusive**
    This allows multiple consumers or workers to process messages from the queue in parallel. This is suitable for scenarios where message order isn't critical and parallel processing is beneficial.
  - **Exclusive**
    This ensures that only one consumer or worker has access to the queue at a time. This is useful in scenarios that require messages to be processed in the order they were received.
- **Number of Concurrent Processes**
  If you select **Non-Exclusive** for the **Access Type**, enter the number of concurrent processes for each worker node. The recommended value depends on the number of worker nodes, the number of queues on the tenant, and the incoming load.
- **Retry Interval**
  Enter a value for the time to wait before retrying message delivery.
- **Exponential Backoff**
  Check the box to double the retry interval after each unsuccessful retry
- **Maximum Retry Interval (in min)**
  If you select **Exponential Backoff**, enter a value for the maximum number of minutes to wait before retrying message delivery.
- **Dead-Letter Queue**
  If you select **Non-Exclusive** for **Access Type**, check this box to take the message out of processing and mark it as Blocked in the queue if message processing has been stopped due to an out-of-memory error in the worker node, the message has later been retried twice by the JMS sender adapter, and each retry has again led to an out-of-memory error in the worker node.

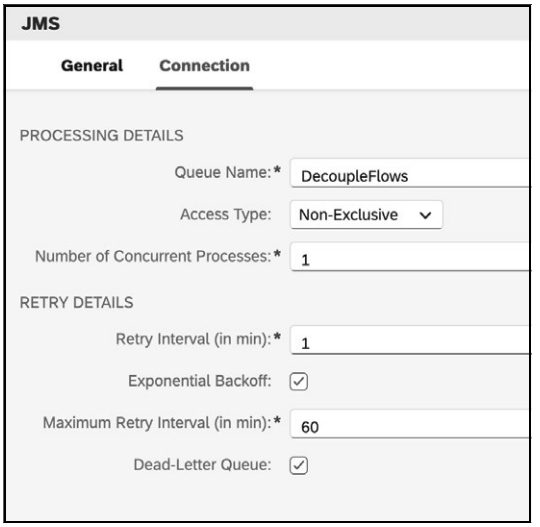

Figure 5.5 JMS Sender Adapter Configuration Properties

Receiver Adapter

In addition to producing messages into a queue, you'll need to consume and process messages from a queue. This is where the receiver adapter comes into play. The following covers how to configure these parameters to alter the processing power of the receiver adapter, which you do on the screen shown in Figure 5.6.

- **Queue Name**
  Enter the name of the message queue.
- **Access Type**
  Select one of the two types of access to the JMS queue:
  - **Non-Exclusive**
    This allows multiple consumers or workers to process messages from the queue in parallel. It's suitable in scenarios where message order isn't critical and parallel processing is beneficial.
  - **Exclusive**
    This ensures that only one consumer or worker has access to the queue at a time. This is useful in scenarios that require messages to be processed in the order they were received.
- **Retention Threshold for Alerting (in d)**
  Enter the time period (in days) by which the messages have to be fetched. The default value is 2.
- **Expiration Period (in d)**
  Enter the time period (in days) by which the messages have to be fetched, which must be no less than the number of days you entered for the **Retention Threshold for**

Alerting. We recommend that you enter a number of days that's at least twice the retention threshold. The default is set to 30 days, and the maximum possible value is 180.

- **Compress Stored Message**
  Check this box to compress the message in the JMS queue, which reduces disk space usage and network traffic.
- **Encrypt Stored Message**
  Check this box to encrypt the message in the JMS queue.
- **Transfer Exchange Properties**
  Check this box to transfer the exchange properties to the JMS queue. We don't recommend using this option because headers and exchange properties are subject to size restrictions, which can result in problems or errors.

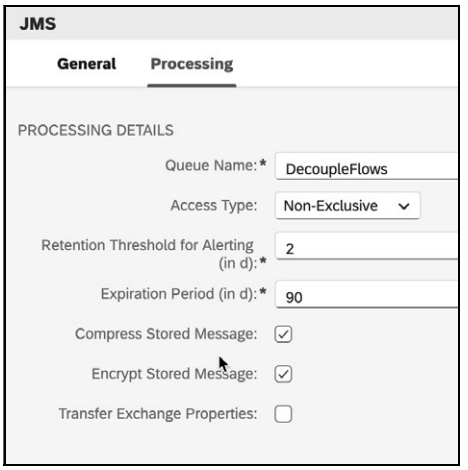

Figure 5.6 JMS Receiver Adapter Configuration Properties

5.3.2 Java Messaging Service Adapter Limitations

Modern integration landscapes demand scalable, fault-tolerant architectures that are capable of handling varying message volumes without downtime or data loss. Asynchronous communication via JMS is one of the cornerstones of such architectures, and in SAP Integration Suite, JMS queues are used to decouple producers and consumers, buffer peak traffic, and recover gracefully from failures.

However, JMS resources are not unlimited. SAP imposes certain quotas and thresholds to maintain system stability, especially in shared cloud environments. You must understanding and optimize these resources to avoid failures, ensure throughput, and build sustainable interfaces.

This section focuses on JMS resource limitations in SAP BTP and strategies for optimizing their usage in production-grade integration scenarios.

Understanding Java Messaging Service Resource Quotas in SAP Business Technology Platform

SAP Integration Suite provides predefined resource quotas for JMS usage, which vary depending on the service plan you select (e.g., standard or advanced). These quotas govern the maximum number of queues, message throughput, concurrent consumers, and overall storage capacity that are available to a tenant. You must understanding these limits so you can plan scalable, high-volume integrations and avoid unexpected failures due to resource exhaustion.

Fortunately, SAP BTP offers self-service entitlement configuration, which allows administrators to request additional capacity through the **Entitlements and Quotas** section of the SAP BTP cockpit. This enables organizations to scale their JMS usage in alignment with growing business needs without requiring custom provisioning or support tickets. Table 5.1 summarizes the typical default values and how they can be expanded. We recommend that you to review these quotas regularly—especially during peak load periods or architectural changes—to ensure sufficient buffer and avoid disruptions in asynchronous message processing.

Resource Types	Standard Plan Limits	Self-Service Maximums
Number of JMS queues	30	100
Queue capacity	9.3 GB	30 GB
Number of transactions	150	500
Number of consumers	150	500
Number of providers	159	500

Table 5.1 JMS Resource Limitations

Message Size Considerations

One of the most critical yet often overlooked limitations in JMS usage within SAP Integration Suite is the size of individual messages. Although there's no fixed, hard limit that's imposed per message, each JMS queue operates under a shared quota that governs the total volume of stored data. This means that a small number of overly large payloads can quickly consume the available capacity, which will leave less room for other messages and potentially cause queues to reach their maximum thresholds. When queues become full, message delivery will fail and potentially disrupt business-critical processes.

To avoid this, you need to implement the following best practices for managing JMS message size:

- Compress payloads if feasible (e.g., JSON → GZIP). Applying compression significantly reduces the byte size of structured data formats like JSON and XML, and this conserves queue space and can improve throughput by reducing transmission time.
- Use external storage for large attachments and include only metadata in the JMS message.
  You should upload documents, images, and binary files to external systems such as SAP Document Management or cloud object storage. The JMS message should then carry a reference link or document ID to minimize in-queue payload weight.
- Keep JMS headers concise since they consume space too. Custom headers are useful for routing and traceability, but you should use them sparingly. Overuse of headers adds overhead and inflates message size.
- Avoid unnecessary XML/JSON verbosity. Trim redundant fields, whitespace, and deeply nested structures to streamline the payload and make better use of JMS quotas.

5.3.3 Transactional Behavior and Concurrency

Each consumer iFlow in Cloud Integration that retrieves messages from a JMS queue operates in transactional mode to ensure reliability and message integrity. In this mode, a message is only considered successfully processed once the entire flow completes without error. If there's a failure of any part of the flow—such as a transformation step, external system call, or routing logic—the transaction is rolled back and the message is returned to the queue for retry. This transactional behavior ensures that no message is lost or partially processed, and that's crucial for maintaining consistency in enterprise data flows.

You can use the following key transactional controls to fine-tune performance and stability:

- **Batch size**
  This defines how many messages are fetched and processed in a single transactional unit. The default batch size is up to 256 messages, and larger batches may lead to processing delays, memory strain, and timeouts. In high-load environments, tuning this parameter helps balance throughput and resource efficiency.
- **Concurrency**
  You can configure the number of parallel threads that process messages simultaneously. Higher concurrency increases processing speed and is ideal for large message volumes, but it also demands more CPU and memory resources. The optimal concurrency setting depends on system capacity and the complexity of the processing logic.
- **Visibility timeout**
  This parameter controls how long a message remains invisible to other consumers once it's picked up. If the flow crashes or hangs during processing, the message

becomes visible again after the timeout so it to be retried. This mechanism prevents message loss in unforeseen failure scenarios.

Together, these controls provide a flexible, high-performance foundation for building resilient, transactional message processing flows.

5.3.4 Queue Design Strategy and Optimization

When you're designing JMS-based messaging architectures in Cloud Integration, you may be tempted to create a dedicated queue for each integration scenario. While this provides clear separation, it can quickly exhaust the limited number of queues that are available per tenant, especially in standard service plans. A more scalable and resource-efficient approach is to use shared queues with internal routing logic inside the iFlows. In this model, messages of different types or use cases are published to the same queue, and the consumer flow applies filters, content-based routing, or header-based selectors to direct each message down the appropriate processing path.

There are several benefits to using shared queue models:

- It reduces total queue count and associated resource consumption. Consolidating use cases onto fewer queues helps them stay within JMS quota limits, leaves room for future growth, and reduces administrative overhead.
- It makes monitoring and management easier. Having fewer queues makes it simpler to track message health, throughput, and failures, so it improves visibility and operational control.
- It facilitates dynamic scaling of consumer flows. With one shared queue, it becomes easier to manage concurrency, tune performance, and respond to spikes in load without adjusting multiple queues individually.

However, this approach requires thoughtful design, and you must still consider message isolation and error handling. For high-risk or business-critical processes where data privacy, compliance, or strict SLA enforcement is required, it may be better to segregate those flows into dedicated queues. This prevents noisy neighbors from interfering with performance, and it simplifies incident resolution for isolated domains. Shared queues, when designed carefully, offer a powerful pattern for efficient and scalable integration design.

5.3.5 Optimization Techniques

Efficient use of JMS queues is a cornerstone of building scalable, resilient, and cost-effective integrations within SAP Integration Suite. As messaging demands grow across systems and business processes, careful management of JMS resources becomes increasingly important. By strategically optimizing queue design, limiting payload

size, and fine-tuning processing patterns such as concurrency and batching, integration teams can ensure high throughput, avoid system bottlenecks, and maintain message integrity. Just as important, these optimizations help organizations stay within defined platform resource limits to prevent unexpected disruptions and enable sustainable growth across the integration landscape. Let's look at some key techniques for accomplishing this.

Limiting Queue Creation

In complex integration landscapes, it's common to see a proliferation of JMS queues—often one per use case or interface. While this offers clean separation, it quickly leads to resource exhaustion and operational overhead. An effective alternative is JMS queue consolidation, in which multiple integration scenarios share a common queue. You can make this possible by unifying the logic of multiple flows into a single, modular iFlow that routes and processes messages dynamically, based on metadata such as JMS headers, message type, or business context.

Using constructs like message routers, content-based filters, and external configuration tables, developers can maintain functional separation within a unified flow. This strategy reduces queue count, optimizes resource usage, and simplifies monitoring, while still maintaining flexibility and extensibility. However, it requires careful design to ensure routing logic is transparent, testable, and maintainable.

Minimizing Message Volume

JMS queues in SAP Integration Suite are subject to overall quota limits, which means excessive message sizes can quickly exhaust available capacity, trigger retries, and degrade performance. To avoid this, you need to keep payloads lean and purpose-fit for message-based communication. One effective strategy is to offload large attachments or bulk data—such as PDFs, images, and data exports—to external storage systems like SAP Document Management service or cloud object stores (e.g., Azure Blob Storage, Amazon S3). The JMS message then only needs to carry metadata or a reference (such as a URL or document ID), and that drastically reduces size while preserving access to full content.

Additionally, content compression techniques, such as applying GZIP to JSON or XML payloads, can further reduce message footprint without sacrificing structure. Implementing size validation early in the iFlow helps reject oversized messages before they consume queue resources. By combining external storage with compression and validation, you ensure that JMS remains a high-performance, cost-effective messaging layer.

Using DLQs Effectively

DLQs are further described in Section 5.4.3, and they're essential components of robust JMS-based integration, particularly for critical business flows where message loss is unac-

ceptable. If you configure a DLQ for such flows, failed messages that exceed retry limits or encounter unrecoverable errors will be automatically redirected for later inspection. This ensures that vital data isn't silently discarded and can be recovered or corrected. To use DLQs effectively, you need to implement a regular monitoring routine—through either SAP's monitoring tools or external alerting systems—to track DLQ growth and identify recurring failure patterns. Periodic manual or automated cleanup and reprocessing should be part of your operational best practices. You can handle reprocessing through a dedicated error-handling iFlow that's capable of enriching or transforming the message before resending it to the original queue or target system. This proactive approach safeguards data integrity, improves system resilience, and provides valuable insights into the long-term quality and stability of the integration landscape.

Scaling Consumers Dynamically

To help you handle fluctuating workloads efficiently, SAP Integration Suite allows for dynamic scaling of JMS consumers, which enables more processing threads during high-traffic periods such as month-end closings, promotional events, and batch imports. By increasing the number of concurrent consumers on a JMS-based iFlow, the system can process more messages in parallel, thus reducing backlog and improving responsiveness. However, indiscriminate scaling can overwhelm downstream systems or saturate memory and CPU resources.

To manage this risk, you should consider implementing *condition-based throttling*, in which the message consumption rate adapts to external triggers such as system availability, queue length, and business hours. You can achieve throttling by using process control parameters, delay patterns, or dynamic configurations fetched from external sources. When combined, consumer scaling and throttling enable iFlows to remain both responsive and controlled, thus ensuring stable performance under pressure without compromising system integrity.

Monitoring Routinely

Consistent monitoring of JMS queues is crucial for maintaining system health and anticipating capacity issues before they impact operations. A well-structured monitoring routine should include weekly utilization checks to track queue depth, processing throughput, and message age, because they help to detect anomalies such as processing delays and stuck messages. Additionally, monthly reporting and forecasting based on historical usage patterns enables teams to predict growth trends, plan for upcoming peak loads, and adjust configurations or upgrade service plans proactively. Combining short-term operational oversight with long-term capacity planning ensures sustainable performance and avoids unexpected service interruptions

5.3.6 Common Pitfalls and Their Mitigation

In this section, we'll walk you through some common pitfalls that organizations may encounter when using a JMS adapter, and we'll show you how to counter them:

- **Too many queues created per use case**
  - **Impact**
    Creating a dedicated JMS queue for each iFlow may initially seem a well-organized process, but it quickly leads to exhaustion of the allowed queue quota, especially in standard SAP BTP service plans. With only about 30 queues available by default, this approach limits scalability and increases administrative overhead.
  - **Mitigation strategy**
    Instead of one queue per interface, you can design shared queues with intelligent message routing. Use JMS header properties (e.g., type, businessArea) and message selectors in the consumer flows to filter and process only relevant messages. This approach reduces the total number of queues required while maintaining logical separation. It also simplifies monitoring and resource allocation to help maintain healthy queue capacity over time.
- **Unbounded payload growth**
  - **Impact**
    Large message payloads consume more space in the queue, reducing available capacity. Since queue retention is shared across all messages, a few large messages can exhaust the queue and prevent new messages from being published. Additionally, large payloads may cause timeouts, retries, and downstream processing delays.
  - **Mitigation strategy**
    To prevent payload-related bottlenecks, adopt a disciplined payload design. Compress data by using algorithms like GZIP, strip unnecessary metadata and whitespace, and consider storing attachments or documents in external repositories (like SAP Document Management or Azure Blob Storage) and referencing them via a link or ID in the message. Also, implement data contracts that enforce message size limits and validation checks early in the producer flow to filter oversized messages before they enter the queue.
- **Overlapping retries due to high concurrency**
  - **Impact**
    Setting high concurrency on a consumer flow can lead to multiple instances trying to process the same message simultaneously, particularly when messages fail and reenter the queue rapidly. This can result in message duplication, increased load, and inconsistent system behavior.
  - **Mitigation strategy**
    Use EOIO configuration if the processing requires strict sequencing or transactional integrity. Limit concurrency where messages must be processed serially to

preserve state consistency. Introduce idempotent logic in your iFlows so that repeated processing doesn't lead to duplicated records or actions. For critical flows, you can use a retry pattern that includes delay and exponential backoff to minimize collision and processing contention.

- **No cleanup of failed DLQ messages**
  - **Impact**
    DLQs accumulate messages that can't be processed due to repeated failures. If these are not monitored and cleaned up, they consume valuable queue space, clutter monitoring dashboards, and may mask real-time issues by obscuring alert visibility.
  - **Mitigation strategy**
    Schedule regular DLQ reviews as part of your operational routine and implement alerting rules when DLQ size exceeds a threshold (e.g., 100 messages). Develop a dedicated error-handling flow to read from DLQs, enrich error messages, and either retry them under modified conditions or escalate them to manual resolution. Document error causes and recurring patterns to improve the robustness of your main processing flows over time.
- **Missing alerts for near-exhausted queues**
  - **Impact**
    Without timely alerts, you may not realize that a queue is nearing its maximum capacity until new messages are rejected. This can lead to business-critical message loss or backpressure on the producing systems, which in turn can cause upstream delays or failures.
  - **Mitigation strategy**
    Proactively configure JMS monitoring alerts in the Integration Suite. Set thresholds for alerting at around 80% usage so you can take action before hard limits are reached. Use the SAP Alert Notification service to forward alerts to operations teams or integrate with external monitoring systems like Splunk or Microsoft Teams. Pair these alerts with automated dashboards that provide visibility into all active queues and their status, so operators can react swiftly to anomalies.

Table 5.2 serves as a quick summary of these common pitfalls, their impacts, and mitigation strategies.

Pitfalls	Impacts	Mitigation Strategies
Too many queues created per use case	They exhaust the queue quota.	Use shared queues with selectors.
Unbounded payload growth	It hits storage limits and increases retries.	Compress and split large payloads.

Table 5.2 JMS Pitfalls and Mitigations

Pitfalls	Impacts	Mitigation Strategies
Overlapping retries, due to high concurrency	Message processing gets duplicated.	Use EOIO and limit concurrency per flow.
No cleanup of failed DLQ messages	It occupies space and increases noise.	Schedule regular DLQ reviews and cleanup routines.
Missing alerts for near-exhausted queues	They cause sudden interface failure.	Set up alert rules in the monitoring dashboard.

Table 5.2 JMS Pitfalls and Mitigations (Cont.)

5.4 Error Handling and Retry Strategies

In asynchronous messaging architectures, error handling and retry strategies are crucial for ensuring reliability and integrity. This is required to keep the entire system and all its components in a consistent state, and it also renders transactional operations completed. Database and other persistent stores need to stay synchronized and may not contain loose, dangling or other inappropriate content. Cloud Integration provides a comprehensive toolkit for managing various error scenarios that arise during message processing.

To address these challenges effectively, you need to understand the different building blocks and strategies that are available in the SAP BTP context. To help you with this, in Section 5.4.1, we explore the exception subprocesses, which provide a structured way to catch and manage errors within iFlows. In Section 5.4.2, we'll look at retry mechanisms, which help ensure that temporary issues don't lead to message loss. In Section 5.4.3, we look at cases where messages can't be processed even after retries and the DLQ offers a safe fallback. In Section 5.4.4, we cover manual message reprocessing, which allows administrators to intervene and resolve issues directly. In Section 5.4.5, we cover error categorization, which makes troubleshooting more effective by helping to distinguish between recoverable and nonrecoverable problems. Finally, Section 5.4.6 covers alerts and monitoring Integration, which ensure that critical issues are surfaced in real time and can be acted upon proactively. Together, these approaches form a holistic framework for building resilient, fault-tolerant integrations with JMS.

5.4.1 Exception Subprocesses

At the core of any resilient integration design lies the principle that failures should be expected and handled deliberately. In Cloud Integration, *exception subprocesses* provide a structured and maintainable way to capture and react to runtime errors during

message processing. Instead of allowing a failure to terminate an iFlow abruptly, an exception subprocess offers a controlled path where the error can be logged, categorized, retried, or rerouted as needed. Developers can define these subprocesses for the entire iFlow or scope them more narrowly around specific steps, such as a content modifier, a message mapping, or an external service call. This flexibility enables developers to differentiate between technical issues—like HTTP timeouts or transformation failures—and business rule violations that may require different handling strategies. Exception subprocesses not only improve system robustness but also provide vital observability, and that makes them foundational building blocks of any effective retry and recovery mechanism in JMS-based integration scenarios.

To implement exception handling effectively, you should focus on granularity, reusability, and traceability. One proven design pattern is to create localized exception subprocesses near the point of failure—for example, wrapping external API calls—so that error handling is contextual and focused. These subprocesses should log essential diagnostic information—such as error type, message ID, timestamp, and custom JMS headers—to support root cause analysis. For recurring logic like error logging or alert generation, you can extract the subprocess into a reusable integration artifact, such as a shared logging flow or centralized monitoring endpoint. Categorization is also key: you should differentiate between transient errors (which are suitable for retry), data-related issues (which are suitable for escalation), and unexpected faults (which are suitable for alerts or manual intervention). You should also avoid implementing generic catch-all subprocesses that simply swallow errors without meaningful action, because that can obscure critical failures and delay response times. By designing exception subprocesses thoughtfully, you as an integration developer can create resilient, maintainable, and transparent error-handling frameworks that strengthen the overall stability and observability of asynchronous messaging flows.

5.4.2 Retry Mechanisms

An essential component of fault-tolerant integration is the ability to recover gracefully from temporary failures without manual intervention. In Cloud Integration, the JMS adapter's built-in *retry mechanisms* are specifically designed to address transient issues such as network instability, temporary backend unavailability, and intermittent timeouts. Rather than immediately marking a message as failed, the adapter pauses and attempts to resend the message based on a preconfigured retry policy. This ensures that message delivery remains reliable without overburdening the source system or requiring real-time availability from downstream systems. The retry logic operates at the queue level, which means the message remains safely in the JMS queue and is only removed upon successful processing. This asynchronous behavior allows for natural buffering during spikes in system load or short-term outages, and that makes the retry mechanism a critical safeguard in loosely coupled architectures.

To configure retries effectively, developers and architects must balance recovery speed with system stability. The most fundamental settings include the **Retry Interval**, which is the pause duration between attempts, and the **Retry Count**, which limits how many times the adapter will attempt delivery before escalating the issue or forwarding the message to a DLQ. For more nuanced control, the adapter also supports *exponential backoff*, which is a strategy that increases the wait time between retries after each failure. This helps keep fragile downstream systems from being overwhelmed while still allowing time for recovery. A well-tuned retry mechanism should align with the availability characteristics of the target system; for example, retrying every five minutes for up to an hour might suit a CRM system that experiences nightly downtime. Additionally, it's important to monitor retry patterns in production: if certain queues frequently hit retry limits, it could signal deeper issues that require architectural changes. Ultimately, when paired with exception subprocesses and proper alerting, JMS retries form a resilient first line of defense against temporary faults that enables seamless and automated recovery within asynchronous integration scenarios.

5.4.3 Dead Letter Queue

In a robust integration landscape, not all errors can or should be resolved through automated retries. In cases where all retry attempts are exhausted, DLQs provide a critical safety net by isolating problematic messages from the main processing path. Instead of letting failed messages block or clog the queue, the system routs them to a designated DLQ for post-mortem analysis and controlled recovery. Each failed message in the DLQ retains important metadata—such as headers, timestamps, error codes, and the original payload—that offers valuable context for debugging. This mechanism not only prevents queue saturation but also ensures that messages requiring manual correction or escalation are retained for later handling. By separating retriable errors from terminal failures, DLQs enable teams to maintain throughput in production environments without sacrificing data visibility or reliability.

From an operational standpoint, effective DLQ management is key to sustainable integration operations. Routine monitoring of DLQ content should be part of your weekly or even daily health checks, particularly for critical business processes. You can manually reprocess messages in the DLQ via the Cloud Integration **Monitoring** UI, either as-is or after applying corrections based on error diagnostics. To streamline this process, integration teams should define escalation thresholds—for example, by triggering an alert when more than 100 messages accumulate or when a specific message type repeatedly fails. These thresholds can trigger alerts via SAP Alert Notification service, or they can be integrated with external ITSM platforms like ServiceNow and Jira. Furthermore, by analyzing recurring error types in DLQs, teams can proactively improve error handling in upstream flows, which can lead to more resilient and self-healing integrations. Also, businesses should never treat DLQs as passive storage—they are powerful operational tools when paired with strong visibility, analytics, and governance strategies. With

well-maintained DLQs in place, businesses can ensure graceful degradation, preserve message integrity, and uphold SLAs, even during failure conditions.

5.4.4 Manual Message Reprocessing

A key advantage of using DLQs in Cloud Integration is the ability to manually reprocess failed messages after root cause analysis or corrective action. This feature ensures that even unrecoverable errors don't result in data loss, provided that appropriate follow-up is performed. You handle reprocessing through the **Monitoring** section of SAP Integration Suite, which offers a user-friendly interface that you can use to review and act on failed transactions. To begin, you navigate to **Monitor • Message Queues** (see Figure 5.7) and open the message queues table. This will bring up a list of configured JMS message queues that you can select, and you can enumerate all messages in the given queue (see Figure 5.8). When a message hasn't been processed successfully, it will show a status of **Failed**, and you can select that message and click **Retry** on the top bar to retry the processing, or you can click **Move** to move the message into a different (e.g., DLQ) queue.

This capability empowers support teams to resolve integration issues efficiently without requiring redeployment or flow modifications. That makes it a powerful tool for operational continuity and SLA compliance.

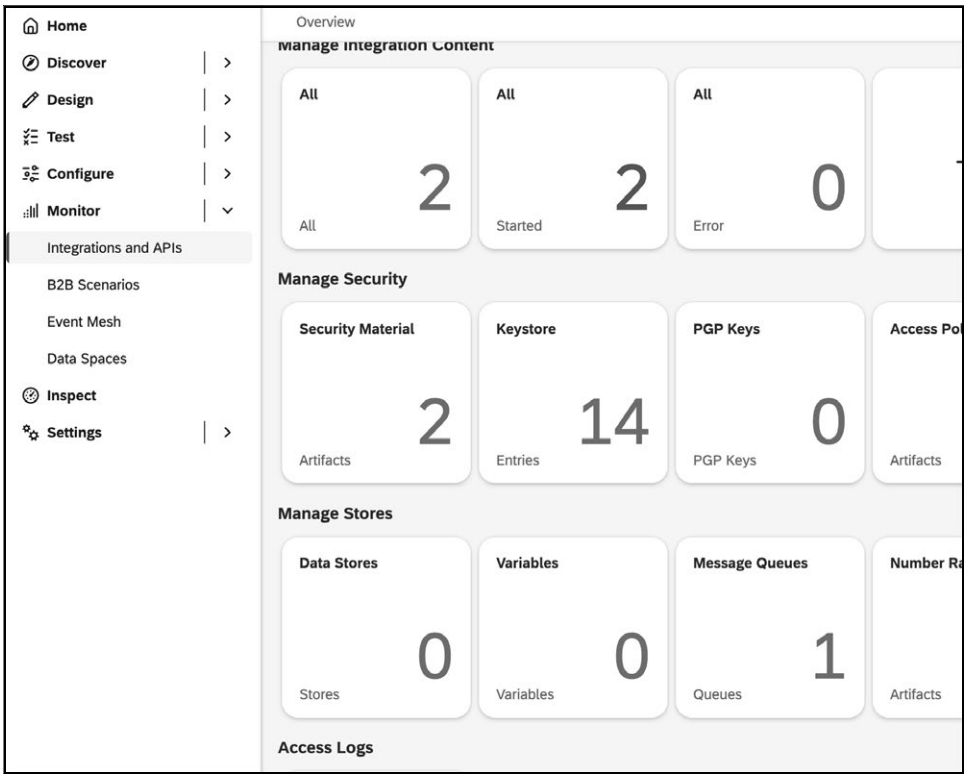

Figure 5.7 Cloud Integration Monitoring Overview

Figure 5.8 Cloud Integration Message Queue Table for Message Management

5.4.5 Error Categorization

Effective error handling in JMS-based iFlows begins with the ability to categorize errors accurately, as not all failures are created equal. Broadly, errors encountered during JMS processing can be classified into three categories: transient, persistent, and system level.

Transient errors are typically temporary issues such as network delays, backend service timeouts, and temporary unavailability of external systems. These are retriable, and the JMS adapter's built-in retry mechanism is well-suited to handle them automatically because it allows the system to recover once conditions normalize.

Persistent errors, on the other hand, stem from data-related problems such as invalid formats, missing fields, and failed validations. These errors are nonretriable and will persist regardless of the number of attempts. In such cases, forwarding the message to a DLQ is the most appropriate response because it enables manual inspection and correction.

System errors are the third and often most critical error category. They include internal integration issues, such as memory exhaustion, configuration errors, code bugs, and unavailable resources like JMS quota overflows. These errors require immediate operational intervention and are typically accompanied by system logs or alerts.

You need to understand which category an error belongs to so you can determine the correct remediation strategy—whether to let the retry mechanism handle it, forward it to a DLQ, or escalate it to the support or infrastructure team. For practical implementation, you should designate flows to tag errors with category-specific identifiers and include detailed exception handling logic that's tailored to each error type. Additionally, having analytics and monitoring tools that report on error trends by category can greatly enhance decision-making and issue triage. Categorization transforms error handling from a reactive to a strategic capability, and that helps organizations build more intelligent, self-aware integrations that maintain operational continuity, even under failure conditions.

5.4.6 Alerts and Monitoring Integration

Proactive monitoring is a cornerstone of operational excellence in enterprise integration, and Cloud Integration provides robust options to support real-time alerting and observability. To detect and react to issues in JMS-based flows, integration developers can leverage native integrations with SAP Alert Notification service, SAP BTP cockpit logs, and third-party external monitoring tools like Splunk, Dynatrace, and Azure Monitor. These integrations allow for seamless tracking of key system events and thresholds, which helps ensure that critical failures or bottlenecks are not only recorded but also immediately actionable. Whether they alert operations teams to an unexpected spike in queue depth or a surge in retry activity, centralized alerts notify teams promptly so they can take corrective action before user-facing impacts occur.

Administrators can define alert triggers based on a range of metrics and thresholds that are tied to JMS queue activity. For example, administrators can configure alerts when message retries exceed a set threshold, which indicates that a downstream system may be unresponsive or overloaded. Similarly, the population of DLQs can trigger a warning, especially if recurring errors begin to accumulate. Another vital trigger is queue size utilization—when a queue approaches 80% or more of its capacity, an alert can prevent service degradation by prompting early investigation. These alerts can be consumed through multiple channels, including email, Slack, Microsoft Teams, and webhook-based integrations with ITSM systems like ServiceNow.

By embedding alerting into your monitoring landscape, you can enable continuous integration health checks, streamline root cause analysis, and enhance the resilience of asynchronous messaging architectures. Ultimately, well-configured alerts not only support faster incident response but also create the transparency needed to uphold SLAs and reduce downtime across business-critical integrations.

5.5 Summary

Loosely coupled interfaces built on JMS queues within SAP BTP offer a proven and powerful architectural approach to designing scalable, fault-tolerant, and resilient enterprise integrations. In increasingly complex and distributed IT landscapes, where system uptime, elasticity, and adaptability are crucial, the ability to decouple producers and consumers of data is no longer a luxury—it's a necessity.

This chapter has demonstrated how asynchronous messaging using JMS forms the foundation for such loosely coupled systems. By removing the direct dependency between sending and receiving systems, asynchronous processing ensures that integrations can continue operating smoothly, even when downstream systems are unavailable, undergoing maintenance, or experiencing performance degradation. The message queue serves as a persistent buffer that enables the system to absorb fluctuations in load, retry transient failures, and continue functioning under unpredictable conditions.

With proper planning and implementation, JMS messaging in SAP BTP empowers organizations to build enterprise-grade integration solutions that are not only reliable and scalable but also flexible, maintainable, and future-proof. As businesses embrace cloud-native design, microservices, and API-first thinking, the role of loosely coupled interfaces becomes increasingly important.

JMS is more than a transport protocol—it's a building block for resilient architecture. By embracing the principles and practices covered in this chapter, integration architects and developers can confidently design systems that respond gracefully to change, recover from failure, and scale to meet the demands of modern enterprise operations.

Koch, Reisinger, Urschick

SAP Cloud Integration Cookbook

Advanced Cloud Integration with SAP Integration Suite

- Find solutions to complex problems, master best practices, and learn advanced Cloud Integration techniques for your SAP system
- Work with Groovy, XSLT, message mappings, adapters, and more
- Create reusable integration artifacts, design loosely coupled interfaces, and manage transports with CI/CD pipelines

www.sap-press.com/6176

The Authors
Martin Koch is the managing director of CloudDNA GmbH, an SAP partner in Austria. Thorsten Reisinger is an integration developer and solution architect at CloudDNA GmbH. Marc Urschick is a development lead at CloudDNA GmbH.

ISBN 978-1-4932-2765-5 • 568 pages • 12/2025
E-book: $84.99 • **Print book:** $89.95 • **Bundle:** $99.99

Rheinwerk Publishing