Entitlement Provisioning Guide

Notifications & Events · https://learn.logisense.com/docs/728460193/

Last verified with: 10.8.6.0

Overview

Provisioning and external system integration patterns determine how LogiSense Billing coordinates commercial changes in the billing platform with operational changes in downstream systems.

That can include:

- activating a service in a network or platform,
- suspending a service when account or service state changes,
- enabling entitlements when an account package becomes active,
- notifying downstream systems that a lifecycle event has occurred,
- or accepting status changes back from an external provisioning platform.

The most important idea is that billing and provisioning do not have to be the same thing.

LogiSense Billing is responsible for the commercial model, lifecycle rules, and entitlement structures. External systems often perform the operational work, such as:

- creating or enabling an external service,
- assigning capacity,
- provisioning access,
- updating a device or subscriber,
- or disabling service when the lifecycle changes.

This guide explains the most common patterns for connecting those two worlds.

The Core Integration Model

At a business level, provisioning integrations usually follow one of two patterns:

- the external system calls the LogiSense API,
- or LogiSense publishes an event and the external system reacts to it.

In practice, many customers use both.

What this means is:

- APIs are typically used for direct command-style actions,
- while webhooks are typically used for event-driven reactions,
- and the strongest integrations often combine both.

Event-Driven Provisioning

Event-driven provisioning means a lifecycle change in LogiSense Billing causes an event to be emitted, and an external system reacts to that event.

Examples include:

- an account package status change,
- an account service state transition,
- a billing activation event,
- or another operational milestone that signals the downstream system should take action.

What this means is:

- LogiSense does not need to directly execute every provisioning step itself,
- the external platform can remain the system that actually provisions the service,
- and the integration can stay decoupled and easier to scale.

This pattern is especially useful when the downstream system:

- owns the actual network or application configuration,
- needs to process changes asynchronously,

- or serves multiple business systems and should not be tightly coupled to one direct API caller.

Why Event-Driven Provisioning Matters

Many provisioning actions are not instantaneous.

They may require:

- workflow orchestration,
- asynchronous processing,
- retries,
- human review,
- or coordination with several downstream services.

An event-driven pattern allows the billing platform to communicate that the business state has changed, while giving the downstream system the freedom to complete the operational work in its own way.

What this means is:

- the billing system can remain cleanly focused on commercial and lifecycle truth,
- and the external system can remain focused on execution and fulfillment.

API Pattern

The API pattern is the direct integration model.

In this model, an external platform or orchestration layer calls LogiSense APIs to:

- create an account package,
- add services,
- schedule an activation,
- update service state,
- suspend or resume service,
- or import usage and related operational data.

What this means is:

- the calling system is explicitly instructing LogiSense to perform a change,
- and the integration behaves more like a command than a notification.

This pattern is a strong fit when:

- another platform owns the order workflow,
- provisioning is tightly orchestrated from an external system,
- or the customer wants deterministic control over when changes are submitted into billing.

Webhook Pattern

The webhook pattern is the outbound event-notification model.

In this model, LogiSense publishes a webhook when a configured event occurs, such as:

- **Account Package Status Change**
- **Account Service State Transition**
- **Billing Activation Rule Achieved**
- and other supported operational or financial events

What this means is:

- LogiSense is notifying an external system that something meaningful has happened,
- and the external system decides what operational action to take next.

This pattern is a strong fit when:

- provisioning should react to lifecycle changes,
- the external system should remain loosely coupled,
- or customers want a reusable event bus style integration model.

API vs Webhook

Pattern	Best used for	Typical direction
API	Direct commands, controlled orchestration, create/update actions	External system to LogiSense

Pattern	Best used for	Typical direction
Webhook	Event notification, asynchronous reaction, downstream provisioning triggers	LogiSense to external system

What this means is:

- use APIs when another system needs to tell LogiSense what to do,
- use webhooks when external systems need to know what has happened in LogiSense,
- and use both when the workflow is two-way.

A Typical Hybrid Pattern

Many customers use a hybrid pattern like this:

1. An order or CRM platform calls LogiSense APIs to create the account package and related services.
2. The package and services are created with the intended lifecycle and pricing structure.
3. LogiSense emits a webhook when the package or service reaches the relevant status or transition.
4. The downstream provisioning platform receives that event and performs the operational enablement.
5. The downstream system may then call LogiSense back through API to confirm, update, suspend, or otherwise synchronize status.

What this means is:

- the integration remains event-driven,
- but still supports explicit command flows where needed.

Retry And Error Handling

Retry and error handling are critical in provisioning integrations because downstream systems are not always available, and asynchronous work does not always succeed on the first attempt.

The platform includes explicit webhook request status concepts such as:

- **Success**

- **Retrying**
- **Failed**
- **Retried**

Webhook request history also tracks details such as:

- request time,
- response status code,
- response body,
- and request duration.

What this means is:

- webhook delivery should be treated as an operational integration process, not just a fire-and-forget notification,
- and customers should design monitoring and exception handling around it.

Recommended Error-Handling Pattern

For customer-facing provisioning design, the strongest pattern is:

1. Treat outbound webhooks as asynchronous requests that can succeed, retry, or fail.
2. Design the receiving system to be idempotent, so the same event can be processed safely more than once.
3. Use webhook request history and integration monitoring to identify failed deliveries or unexpected responses.
4. Keep provisioning state transitions explicit, so retries do not create ambiguous service state in either system.
5. Where confirmation is important, let the downstream system update LogiSense through API or lifecycle state once the operational action is complete.

What this means is:

- reliable provisioning comes from both delivery retry behavior and good downstream design,
- not from assuming every call succeeds once.

Sample Activation Flow

One common pattern is activation after package creation.

Example Flow

1. An order platform creates an account package in LogiSense through API.
2. The package creates its account services and associated operational structures.
3. If the lifecycle moves into the configured activation state, LogiSense emits the relevant lifecycle event.
4. A webhook notifies the external provisioning system that activation should occur.
5. The provisioning system enables the external service.
6. If needed, the provisioning platform updates LogiSense with the resulting lifecycle state or service status.

What this means is:

- LogiSense controls when the commercial lifecycle says a service should be active,
- and the downstream system performs the real-world enablement work.

Sample Suspend Flow

Suspension is usually the inverse pattern.

Example Flow

1. A service or account package moves into a suspended state in LogiSense.
2. A webhook is emitted for the package status change or service state transition.
3. The downstream provisioning system receives the event.
4. The external platform disables, pauses, or restricts service operationally.
5. If additional status synchronization is required, the external system updates LogiSense through API.

What this means is:

- the billing platform and provisioning platform can stay aligned on lifecycle state,
- while still allowing each to perform its own role.

Enabling Entitlements On Account Package Activation

One of the most important provisioning-related behaviors is that entitlements are not just abstract product promises.

When an account package is created from the catalog, the workflow creates the related account service and bucket structures needed to operationalize included usage and billing activation behavior.

What this means is:

- the entitlement model is established as part of account package and account service creation,
- and package activation can act as the lifecycle point at which those entitlements become meaningful for billing and consumption.

In practical terms, customers should think of account package activation as the point where:

- the subscribed offer becomes live,
- the included service structure exists,
- and the corresponding entitlement or bucket logic is ready to govern usage.

Why This Matters For Entitlements

If an offer includes:

- included usage,
- shared allowances,
- activation-triggered service access,
- or billing activation bucket behavior,

then the provisioning design should account for the fact that commercial enablement and operational enablement need to stay aligned.

What this means is:

- if the package is active but the external system is not provisioned, the customer may have a mismatch between commercial and operational state,
- and if the external service is enabled but the package is not in the right lifecycle state, billing and entitlement behavior may not match the intended offer.

Using State-Based Transitions During Provisioning

State-based transitions are one of the cleanest ways to integrate billing lifecycle and provisioning lifecycle.

Instead of treating activation, suspension, resumption, and cancellation as vague operational actions, the platform lets them be modeled as explicit service state transitions such as:

- **Pre-Active to Active**
- **Active to Suspended**
- **Suspended to Active**
- **Active to Cancelled**

What this means is:

- provisioning can be triggered from a specific lifecycle event,
- billing behavior can change at the same point,
- and one-time transition charges can also be aligned to that same operational step.

This is much cleaner than relying on custom flags or ad hoc status logic.

Why State-Based Provisioning Is Powerful

When provisioning uses state transitions as the source of truth:

- the external system knows exactly what lifecycle move occurred,
- the billing system knows exactly what recurring and transition pricing should apply,
- and both sides can reason about the same commercial event.

This is especially useful for:

- activation fees,
- suspension and restoration flows,
- staged onboarding,
- trial-to-paid conversion,
- and operational workflows that should map cleanly to billable lifecycle changes.

Sample State-Based Activation Pattern

Example

1. A service is initially created in **Pre-Active**.
2. The downstream system completes provisioning checks.
3. The service transitions from **Pre-Active** to **Active**.
4. LogiSense recognizes the state transition.
5. Recurring billing now follows the **Active** state pricing.
6. If configured, a one-time activation transition charge also applies.
7. A webhook can notify external systems that the activation transition has occurred.

What this means is:

- one lifecycle transition can drive both the operational and financial outcome,
- instead of requiring separate disconnected logic.

Sample State-Based Suspend Pattern

Example

1. A service is in **Active**.
2. A non-payment rule, customer request, or operational process moves it to **Suspended**.
3. LogiSense recognizes the state transition and applies the suspended-state pricing behavior.
4. A webhook notifies the external platform that operational suspension should occur.
5. If the customer is later restored, the service transitions from **Suspended** back to **Active** and can trigger reactivation logic.

What this means is:

- the suspend and restore process becomes traceable, billable, and operationally consistent.

SaaS Example

A SaaS provider may:

- create a package and services through API,
- keep the service in **Pre-Active** until provisioning finishes,
- then transition the service to **Active** once the tenant, environment, or feature access is enabled.

Why this works:

- the commercial lifecycle stays explicit,
- entitlements are tied to the live package structure,
- and downstream automation can react to status-change webhooks instead of polling.

Telecom Example

A telecom provider may:

- create a subscriber package in LogiSense,
- provision the SIM or line in an external network system,
- use service state transitions for activation and suspension,
- and notify external orchestration through **Account Service State Transition** webhooks.

Why this works:

- the provider can align billing, suspension rules, and operational line state,
- and can also apply activation or reactivation pricing at the same lifecycle moments.

Best-Practice Pattern

The strongest general pattern is:

1. Use APIs for create and update commands.

2. Use webhooks for event-driven downstream provisioning.
3. Use explicit service states and transitions as the lifecycle source of truth.
4. Keep entitlement activation aligned to package and service activation.
5. Design receiving systems to handle retries safely and idempotently.

What this means is:

- provisioning becomes more reliable,
- billing and operations stay aligned,
- and the integration can scale without turning into a brittle point-to-point workflow.

Final Takeaway

Provisioning integrations work best when LogiSense Billing is treated as the source of commercial lifecycle truth, while external systems are treated as the executors of operational fulfillment.

APIs are how external systems tell LogiSense what to do. Webhooks are how LogiSense tells external systems what has happened. State transitions provide the shared lifecycle language between them. And package activation plus account-service bucket creation ensure that entitlements become real operational billing structures rather than just product descriptions.

When those patterns are combined well, customers get a cleaner, more reliable provisioning architecture with stronger alignment between pricing, lifecycle, and service delivery.