RevenueOS: The Implementation Series

ROS-01 · Module 1 · FOUNDATION

# Product Catalog and Pricing Setup

*Designing and governing the commercial foundation — products, pricing models, bundles, discounts, credits, catalog versioning, and pricing approval workflows*

***Get the catalog right. Everything downstream depends on it.***

*Vendor-agnostic specification · For all vendor comparisons see ROS-23*

PREFACE

## Get the Catalog Right. Everything Downstream Depends on It.

*The commercial foundation before the first customer is acquired.*

Before a customer can be billed, before a quote can be generated, before a revenue recognition entry can be posted, someone must define what the company is selling, at what price, and under what rules. That definition lives in the product catalog.

The product catalog is not a marketing list. It is a commercial data structure — a set of objects with defined schemas, relationships, and rules that govern how every downstream system (CPQ, billing, entitlements, revenue recognition) interprets the commercial offer. A product catalog that was designed informally, extended opportunistically, and never cleaned up is the single most common root cause of billing

inaccuracies, CPQ errors, and revenue recognition findings in fast-growing AI companies.

This book specifies the product catalog from the ground up: how to model products and pricing plans, how to configure AI-era pricing models (token packages, outcome components, agent task credits), how to design bundles with defensible allocation methodologies, how to govern discounts and credits without leakage, and how to version and change the catalog without breaking existing customer relationships.

Three principles run through every chapter:

Data first. The catalog is a data architecture problem before it is a user experience problem or a process problem. Get the object schemas right, define the business rules precisely, and the processes and interfaces will follow. Skip the data architecture and the processes will continuously work around it.

Rules are not exceptions. Every time a billing engineer writes ad-hoc logic to handle a special case, a business rule has been missed in the catalog design. The goal of this book is to name every rule explicitly — so that the implementation captures the intended behavior without custom code for every edge case.

Versioning is not optional. The catalog will change. Prices will increase. Products will be deprecated. New AI capabilities will require new pricing models. A catalog without a versioning architecture is a catalog that cannot change safely — and a catalog that cannot change safely will be changed unsafely, with consequences that surface months later in billing disputes and audit findings.

CHAPTER 1.1

# Product Catalog Architecture: Objects, Hierarchy, and Pricing Layer

## Learning Objectives

• Understand the canonical data model for AI product catalogs and how the five core objects relate to each other

• Define the product hierarchy appropriate for your company's catalog complexity

• Specify the catalog state machine and the valid transitions between states

• Identify the minimum data requirements for each object before it can be used in a quote or billing run

• Design the pricing layer separation that allows pricing to change without changing product definitions

## Deployment Context Guidance

| AI-NATIVE SaaS | TRANSITIONING ENTERPRISE | LARGE INTERNAL PLATFORM |
|---|---|---|
| Start with the simplest hierarchy that captures your pricing complexity. Resist the temptation to build a six-level hierarchy for a catalog with eight products. Add levels only when a genuine grouping need arises. | Map your existing product table to the canonical objects before attempting a migration. The most common failure: treating legacy SKUs as canonical product IDs and inheriting all the naming inconsistencies and missing fields that come with them. | Governance of the product hierarchy requires a defined catalog owner — typically the Revenue Operations or Pricing team. Without ownership, every business unit will add products and pricing plans in whatever format is convenient, and the catalog will fragment within 12 months. |

## Process Flow

1. Product Family definition: identify the highest-level groupings that your go-to-market and finance teams use to organize the portfolio. These are the top of the hierarchy — typically 2–5 families for most AI companies.

2. Product definition within each family: each discrete sellable capability or subscription unit is a Product. Products have names, descriptions, AI layer classification (compute / model / token / agent / outcome), and metadata required for catalog search and filtering.

3. Pricing Plan attachment: each Product has one or more Pricing Plans. A Pricing Plan represents a specific commercial structure — flat monthly, per-seat, token-based, outcome-based — attached to a Product for a specific effective date range.

4. Price object within each Pricing Plan: the Price object contains the actual monetary values, unit definitions, tier thresholds, overage rates, and trial configurations. The Price is the leaf node of the catalog hierarchy.

5. Bundle assembly (optional): Products can be combined into Bundles. A Bundle is itself a Product — it can have its own Pricing Plan and Price objects that are separate from the component products' standalone prices.

6. Catalog publication: when the catalog is ready for quoting and billing, it is published. Published catalog items are immutable — changes require a new version or a new item.

## Data Model

```
OBJECT: ProductFamily
  id                UUID          REQUIRED    Unique identifier
  name              string        REQUIRED    Display name (max 100 chars)
  description       string        OPTIONAL    Internal description
  status            enum          REQUIRED    ACTIVE | INACTIVE
  created_at        ISO 8601 UTC  SYSTEM
  updated_at        ISO 8601 UTC  SYSTEM

OBJECT: Product
  id                UUID          REQUIRED
  family_id         UUID          REQUIRED    FK → ProductFamily.id
  name              string        REQUIRED    Display name (max 200 chars)
  description       string        OPTIONAL
  sku               string        REQUIRED    Internal SKU; unique within
company; immutable once set
  ai_layer          enum          REQUIRED    COMPUTE | MODEL | TOKEN | AGENT |
OUTCOME | PLATFORM
  product_type      enum          REQUIRED    STANDALONE | BUNDLE | ADDON |
COMPONENT
  status            enum          REQUIRED    DRAFT | ACTIVE | DEPRECATED |
ARCHIVED
  is_standalone_sellable  boolean REQUIRED    Can this product appear on a quote
without being in a bundle?
  tags              string[]      OPTIONAL    Free-form tags for filtering
  metadata          object        OPTIONAL    Extensible key-value metadata
  created_at        ISO 8601 UTC  SYSTEM
  updated_at        ISO 8601 UTC  SYSTEM

OBJECT: PricingPlan
  id                UUID          REQUIRED
  product_id        UUID          REQUIRED    FK → Product.id
  name              string        REQUIRED    e.g. "Standard Monthly 2025" —
human-readable
  pricing_model     enum          REQUIRED    FLAT | PER_SEAT | TIERED | VOLUME |
USAGE | TOKEN | OUTCOME | HYBRID
  currency          ISO 4217      REQUIRED    e.g. USD, EUR, GBP
  billing_period    enum          REQUIRED    MONTHLY | QUARTERLY | ANNUAL |
ONE_TIME | USAGE
  effective_from    ISO 8601 date REQUIRED
  effective_to      ISO 8601 date OPTIONAL    null = no end date
  status            enum          REQUIRED    DRAFT | ACTIVE | DEPRECATED
  trial_enabled     boolean       REQUIRED    default false
  trial_days        integer       CONDITIONAL Required when trial_enabled = true;
min 1, max 365
  created_at        ISO 8601 UTC  SYSTEM
  updated_at        ISO 8601 UTC  SYSTEM

OBJECT: Price
  id                UUID          REQUIRED
  pricing_plan_id   UUID          REQUIRED    FK → PricingPlan.id
  unit_amount       integer       REQUIRED    Amount in smallest currency unit
(cents). Min 0.
```

```
  unit_type          string        REQUIRED    What one unit represents: "user",
"token", "task", "gb", "outcome"
  min_quantity       integer       OPTIONAL    Minimum purchasable quantity;
default 1
  max_quantity       integer       OPTIONAL    Maximum purchasable quantity; null
= no limit
  tiers              Tier[]        CONDITIONAL Required when
PricingPlan.pricing_model = TIERED or VOLUME
  overage_unit_amount integer      CONDITIONAL Required when
PricingPlan.pricing_model = USAGE or TOKEN
  floor_amount       integer       OPTIONAL    Minimum charge per billing period
regardless of usage (cents)
  ceiling_amount     integer       OPTIONAL    Maximum charge per billing period
(cents)
  created_at         ISO 8601 UTC  SYSTEM

OBJECT: Tier
  tier_index         integer       REQUIRED    Sequence (1-based)
  from_quantity      integer       REQUIRED    Inclusive lower bound of this tier
  to_quantity        integer       OPTIONAL    Inclusive upper bound; null =
unlimited
  unit_amount        integer       REQUIRED    Per-unit amount for this tier
(cents)
  flat_fee           integer       OPTIONAL    Fixed fee applied when this tier is
reached (cents)
```

| BUSINESS RULES |
|---|
| **CAT-001:** A Product must be in status ACTIVE and have at least one active PricingPlan before it can appear on a Quote. |
| **CAT-002:** A Product in status DEPRECATED cannot be added to new Quotes but continues to govern existing Contracts. Its PricingPlan effective_to date governs when billing stops. |
| **CAT-003:** A Product in status ARCHIVED cannot appear in any new commercial context — not on quotes, not as a bundle component, not as an entitlement. It is a historical record only. |
| **CAT-004:** PricingPlan.pricing_model must be consistent with the Product.ai_layer. TOKEN pricing model requires ai_layer = TOKEN or AGENT. OUTCOME pricing model requires ai_layer = OUTCOME. |
| **CAT-005:** A PricingPlan.effective_from date cannot be in the past at time of creation unless the user has PRICING_ADMIN role and provides a business justification. |
| **CAT-006:** Two active PricingPlans for the same Product cannot have overlapping effective date ranges in the same currency. |
| **CAT-007:** Price.unit_amount must be a non-negative integer. A $0 unit_amount is permitted (for free-tier products) but requires a business justification comment. |
| **CAT-008:** Price.floor_amount, if set, must be ≥ the minimum expected charge at Price.min_quantity. |

## API Specification

```
POST   /catalog/products
       Auth: API key (CATALOG_WRITE scope)
       Request: ProductFamily.id, name, sku, ai_layer, product_type,
is_standalone_sellable, [description, tags, metadata]
       Response: 201 Created — Product object
       Errors: 400 (validation failure, duplicate SKU), 403 (insufficient scope),
422 (invalid ai_layer value)

GET    /catalog/products/{id}
       Auth: API key (CATALOG_READ scope)
       Response: 200 — Product object with embedded PricingPlans
       Errors: 404 (not found), 403

PATCH  /catalog/products/{id}
       Auth: API key (CATALOG_WRITE scope)
       Request: Any mutable field (name, description, tags, status). SKU is
immutable.
       Response: 200 — updated Product object
       Errors: 400, 403, 404, 409 (status transition not permitted — e.g.,
ARCHIVED → ACTIVE)

GET    /catalog/products
       Auth: API key (CATALOG_READ scope)
       Query params: status, ai_layer, family_id, page, page_size
       Response: 200 — paginated Product list
       Errors: 400 (invalid filter)

POST   /catalog/pricing-plans
       Auth: API key (CATALOG_WRITE scope)
       Request: product_id, name, pricing_model, currency, billing_period,
effective_from, [effective_to, trial_enabled, trial_days]
       Response: 201 — PricingPlan object
       Errors: 400, 403, 409 (date range overlap for same product/currency)

POST   /catalog/prices
       Auth: API key (CATALOG_WRITE scope)
       Request: pricing_plan_id, unit_amount, unit_type, [min_quantity,
max_quantity, tiers, overage_unit_amount, floor_amount, ceiling_amount]
       Response: 201 — Price object
       Errors: 400 (tiers required for TIERED model, missing overage for USAGE),
403
```

## Exception Handling and Edge Cases

**1. Product deprecated mid-contract: A customer has a 3-year contract for a product now being deprecated. The product must remain in status DEPRECATED (not ARCHIVED) for the contract duration. The billing system must continue to resolve the deprecated product's pricing plan for renewals and amendments. The CPQ system must prevent the deprecated product from appearing in new quote templates but must permit it on renewal quotes generated from existing contracts.**

**2. Pricing plan change after quote exists but before contract signed: A Quote has been generated with PricingPlan v2. The pricing team activates PricingPlan v3 (higher prices) before the customer signs. Business rule: the quote remains valid at PricingPlan v2 pricing until the quote expiry date. After quote expiry, a new quote must be generated and will use the current active pricing plan. The system must not retroactively update signed quotes.**

**3. Bundle contains component being deprecated: A Bundle includes Component A, which is being deprecated. If Component A is deprecated while the Bundle is still actively sold, the Bundle must be reviewed and either: (a) Component A is replaced with a successor product, (b) the Bundle is deprecated, or (c) an exception is approved by the catalog owner. Automated alerts must fire when any bundle component changes status to DEPRECATED.**

**4. Currency mismatch in multi-entity deals: A customer with subsidiaries in the US and EU wants a single deal with USD pricing for the US entity and EUR pricing for the EU entity. PricingPlans are currency-specific — the system must support a single Product with pricing plans in multiple currencies, and the deal desk must be able to combine currency-specific line items in a single Quote.**

**5. Zero-amount product requiring justification: A $0 product (free tier or internal use) must have a documented business justification in the catalog. Without justification, the billing system cannot distinguish intentional $0 pricing from a configuration error. The justification field must be non-null for any Price with unit_amount = 0.**

## Integration Checklist

□ □ Product object is successfully created via POST /catalog/products with all required fields □ SKU uniqueness constraint is enforced — duplicate SKU returns 400, not silent dedup □ PricingPlan creation returns 409 when date ranges overlap for same product/currency □ CAT-006 (no overlapping active pricing plans) is enforced for all currency/product combinations □ Product status transitions are enforced: DRAFT → ACTIVE is permitted; ARCHIVED → ACTIVE is rejected □ Deprecated products appear in GET /catalog/products?status=DEPRECATED □ Archived products do NOT appear in catalog search unless explicitly filtered with status=ARCHIVED □ PricingPlan with pricing_model=TIERED requires at least two Tier objects — validated on creation □ Price.unit_amount = 0 requires justification field populated — validated on creation □ Bundle creation validates that all component product IDs exist and are in ACTIVE status □ Catalog change events are emitted to the event bus for downstream system updates (CPQ, billing engine, entitlement system) □ Event schema for product.updated and pricing_plan.activated contains all fields required by downstream consumers □ Access control: CATALOG_READ scope cannot modify objects;

CATALOG_WRITE scope is required for all mutations □ Audit log captures: who created/modified/deprecated each object, timestamp, before/after state

| CHAPTER 1.1 — Key Takeaways |
| --- |
| › The catalog is a data architecture problem before it is a UX problem — get the object schemas right first |
| › Five core objects: ProductFamily → Product → PricingPlan → Price → (optional) Bundle |
| › The pricing layer is separated from the product layer — prices change frequently; product definitions should not |
| › The catalog state machine governs what can happen at each status — DRAFT products cannot be quoted, ARCHIVED products cannot be referenced |
| › Every catalog object emits an event on mutation — downstream systems (CPQ, billing, entitlements) subscribe to these events rather than polling |

CHAPTER 1.2

# Pricing Models for AI Products: Flat, Per-Seat, Usage, Token, Outcome, Hybrid

## Learning Objectives

• Map each of the eight pricing model types to the AI product scenarios where each is appropriate

• Configure the data model for each pricing model type with the correct required and conditional fields

• Define billable units precisely for USAGE, TOKEN, and OUTCOME pricing models

• Specify tier structures for TIERED and VOLUME models with no gaps and no overlaps

• Identify the revenue recognition treatment for each pricing model and flag RevRec implications for the HYBRID model

## Process Flow

1. Model type selection: based on the product's AI layer, the customer's ability to predict usage, and the company's revenue recognition requirements, select the primary pricing model from the eight supported types.

2. Unit definition: define precisely what constitutes one billable unit. For usage pricing: what event type triggers a consumption count? For token pricing: which token types are included (input only, output only, cached, extended thinking)? For outcome pricing: what exactly constitutes a verified outcome?
3. Tier and threshold configuration: for tiered and volume models, define the quantity boundaries and per-unit rates for each tier. Verify there are no gaps (quantities not covered by any tier) or overlaps.
4. Overage rule configuration: for consumption models with an allocation ceiling, define what happens when the customer exceeds the included allocation. Options: hard stop (deny), soft stop (allow + alert), overage charge (allow + bill at overage rate).
5. Trial configuration: if the product has a free trial period, configure the trial duration, the trial allocation (if applicable), and the trigger for trial-to-paid conversion.
6. Price floor and ceiling: optionally set a minimum charge (floor) regardless of consumption, and a maximum charge (ceiling) regardless of consumption. Both are expressed in cents per billing period.

## Data Model

```
PRICING MODEL TYPES — Complete Specification:

FLAT
  Description: Fixed charge per billing period regardless of usage
  Required fields: unit_amount (the flat fee), billing_period
  Optional fields: min_quantity (default 1), max_quantity
  Revenue recognition: Ratable over the billing period
  AI product fit: Platform access fees, base subscription for AI features with
separate usage billing

PER_SEAT
  Description: Charge per named user, per AI agent slot, or per deployed instance
  Required fields: unit_amount (per seat), unit_type (user / agent / instance),
billing_period
  Optional fields: min_seats (minimum commit), max_seats (soft ceiling requiring
approval above)
  Revenue recognition: Ratable over the billing period; seat changes prorate
within period
  AI product fit: AI-assisted tools where per-human-user access is the primary
metric; agent deployment slots

TIERED (graduated)
  Description: Per-unit rate changes as quantity crosses tier boundaries; each
tier rate applies only to units in that tier
  Required fields: tiers (at least 2, no gaps, last tier to_quantity = null),
billing_period
  Optional fields: floor_amount
  Revenue recognition: Ratable; usage tiers recognized as consumed
  AI product fit: API access with declining per-call rates at volume

VOLUME (flat tier)
  Description: All units priced at the rate corresponding to the total quantity
```

```
consumed; one rate applies to all units
  Required fields: tiers (at least 2, no gaps), billing_period
  Optional fields: floor_amount
  Revenue recognition: Recognized as consumed within period
  AI product fit: Bulk token purchases where all tokens in the period are at a
single rate

USAGE (metered, postpaid)
  Description: Charge based on actual consumption measured by metering events;
billed in arrears
  Required fields: unit_amount, unit_type, event_type (metering event name)
  Optional fields: billing_period (default MONTHLY), overage_unit_amount (if
separate from unit_amount), floor_amount, ceiling_amount
  Revenue recognition: Recognized as consumed; variable consideration if ceiling
applies
  AI product fit: Pure API-call pricing; inference compute pricing

TOKEN (prepaid drawdown)
  Description: Customer pre-purchases a token allocation; tokens are drawn down
as consumed; over-allocation triggers top-up or hard stop
  Required fields: token_types[] (which token types this plan covers),
token_unit_cost (cost per token in billable cents), allocation_per_period (tokens
included per billing period)
  Optional fields: overage_handling (HARD_STOP | SOFT_STOP | CHARGE),
overage_rate (cents per additional token), rollover_enabled (boolean),
rollover_cap_tokens
  Revenue recognition: Prepaid allocation creates deferred revenue; recognized as
tokens consumed; unused + expired tokens recognized at expiry
  AI product fit: Core AI model access (see ROS-07 Token Factory for full
specification)

OUTCOME (pay-per-verified-result)
  Description: Customer pays per verified business outcome delivered by the AI
  Required fields: outcome_definition_id (FK → OutcomeDefinition), unit_amount
(per verified outcome), verification_method (CUSTOMER_ACCEPTANCE |
THIRD_PARTY_AUDIT | AUTOMATED_SIGNAL)
  Optional fields: quality_tier (for tiered outcome quality pricing),
sla_penalty_rate (SLA credit formula)
  Revenue recognition: Recognized at outcome verification event; variable
consideration analysis required
  AI product fit: Contract review AI (per reviewed document), customer service AI
(per resolved ticket), clinical documentation AI (per accepted note)

HYBRID
  Description: Combination of two or more pricing models on a single product;
most commonly a FLAT subscription floor plus USAGE consumption
  Required fields: components[] (array of 2+ pricing components, each with its
own pricing_model and configuration)
  Optional fields: component_allocation_method (how revenue is allocated across
components)
  Revenue recognition: Each component has its own performance obligation;
allocation methodology required
  AI product fit: Subscription floor + token overage; platform fee + per-outcome
billing
```

| BUSINESS RULES |
| --- |
| **PRICE-001:** USAGE and TOKEN pricing models require a defined event_type that maps to a metering event schema. The event_type must exist in the metering system's event registry before the pricing plan can be activated. |
| **PRICE-002:** OUTCOME pricing requires an OutcomeDefinition object with a verified verification_method before activation. An OutcomeDefinition with verification_method = null is invalid. |
| **PRICE-003:** TIERED and VOLUME pricing require at least two tiers. The last tier must have to_quantity = null (no upper bound). There must be no gap between consecutive tiers (Tier[n].to_quantity + 1 = Tier[n+1].from_quantity). |
| **PRICE-004:** HYBRID pricing must specify component_allocation_method. If not specified, the system defaults to RELATIVE_FAIR_VALUE, which requires SSP (standalone selling price) documentation for each component. |
| **PRICE-005:** For all pricing models with a floor_amount, the floor applies to the full billing period regardless of mid-period upgrades or downgrades. |
| **PRICE-006:** ceiling_amount, if set, creates a variable consideration situation under ASC 606. The RevRec system must be notified when a ceiling is added to any pricing plan. |
| **PRICE-007:** trial_enabled = true requires trial_days > 0 and a defined trial-to-paid conversion trigger. Trial conversion is not automatic — the trigger must be explicitly configured (date expiry, usage threshold, or explicit customer action). |

## API Specification

```
POST   /catalog/pricing-plans/{id}/tiers
       Auth: CATALOG_WRITE
       Request: Array of { tier_index, from_quantity, to_quantity, unit_amount,
[flat_fee] }
       Response: 201 — updated PricingPlan with tiers
       Errors: 400 (gap between tiers, last tier has to_quantity ≠ null, fewer
than 2 tiers), 403

POST   /catalog/pricing-plans/{id}/outcome-definition
       Auth: CATALOG_WRITE
       Request: { definition_text, verification_method,
verification_signal_schema, quality_tiers[] }
       Response: 201 — OutcomeDefinition object
       Errors: 400 (verification_method null), 403, 404 (pricing plan not found)

GET    /catalog/pricing-models
       Auth: CATALOG_READ
       Response: 200 — array of pricing model type definitions with
required/optional field lists

POST   /catalog/pricing-plans/{id}/trial-config
       Auth: CATALOG_WRITE
       Request: { trial_days, trial_allocation (for TOKEN plans),
```

```
conversion_trigger_type, [conversion_trigger_value] }
       Response: 200 — updated PricingPlan
       Errors: 400 (trial_days ≤ 0, missing conversion_trigger_type)
```

## Integration Checklist

- ☐ All eight pricing model types are validated — each model type has the correct required fields populated
- ☐ TIERED models have at least 2 tiers with no gaps (validated by API on creation)
- ☐ USAGE models have a registered event_type in the metering event registry
- ☐ OUTCOME models have an approved OutcomeDefinition with a non-null verification_method
- ☐ TOKEN models have overage_handling configured when included_tokens is set
- ☐ HYBRID models have component_allocation_method specified and SSP documentation exists
- ☐ All pricing models with floor_amount or ceiling_amount have been reviewed by RevRec team

**CHAPTER 1.2 — Key Takeaways**

› Eight pricing model types cover all AI commercial scenarios from flat subscription through outcome-based billing

› The TOKEN pricing model is the foundation of AI credit economics — see ROS-07 (Token Factory) for the complete credit lifecycle specification

› OUTCOME pricing requires an OutcomeDefinition before activation — the outcome definition is the billing contract

› HYBRID pricing creates multiple performance obligations under ASC 606 — RevRec review is mandatory

› ceiling_amount on any pricing plan creates variable consideration — automatic RevRec flag on creation

CHAPTER 1.3

# AI-Era Pricing: Token Packages, Agent Task Credits, and Outcome Components

## Learning Objectives

- Design token product configurations with the correct token type definitions and cost structure
- Specify agent task credit economics including task completion definitions and timeout handling
- Configure outcome components with verifiable outcome definitions and appropriate verification methods
- Assemble hybrid AI products from component pricing models with the correct allocation methodology
- Apply the specific business rules governing AI-era pricing that differ from standard SaaS pricing

## Process Flow

1. Token product design: define the token product's credit economy — how many token types are supported, what each type represents (input, output, cached, extended thinking), what the relative cost ratios are between types, and what the exchange rate is from tokens to dollars.
2. Token credit pack structure: design the commercial packaging of tokens for customer purchase. A credit pack is a specific quantity of tokens at a specific price — it is a Price object with pricing_model = TOKEN.
3. Multi-wallet model: if different token types are priced differently and require separate balance tracking (e.g., standard tokens vs extended thinking tokens), define the wallet structure. See ROS-07 (Token Factory) for the complete wallet specification.
4. Agent task credit design: agent task pricing charges per discrete agent task completion rather than per token consumed. Define what constitutes a completed task, how task complexity affects pricing (flat per task vs tiered by task type), and how task credits relate to underlying token consumption (if the agent uses tokens internally).
5. Outcome component definition: for products delivering outcome-based pricing, define the OutcomeDefinition — what the outcome is, how it is verified, what quality tiers exist (if applicable), and how SLA credits are calculated when outcomes fall below committed rates.
6. Hybrid assembly: combine any of the above into a hybrid product. The most common AI hybrid: a subscription floor (monthly access fee) plus token consumption overage. Specify the component allocation methodology for revenue recognition.

## Data Model

```
OBJECT: TokenProductConfig (extends Price for pricing_model = TOKEN)
  token_types           TokenType[]   REQUIRED    List of token types this plan
covers
  input_token_cost      integer       REQUIRED    Cost per 1M input tokens
(cents)
  output_token_cost     integer       REQUIRED    Cost per 1M output tokens
(cents)
  cached_token_cost     integer       OPTIONAL    Cost per 1M cached tokens
```

```
(typically lower); default = input_token_cost
  extended_thinking_cost integer      OPTIONAL    Cost per 1M extended thinking
tokens; default = output_token_cost
  included_tokens       integer       OPTIONAL    Tokens included per billing
period before overage; null = no inclusion
  overage_model         enum          CONDITIONAL HARD_STOP | SOFT_STOP | CHARGE;
required when included_tokens is set
  rollover_enabled      boolean       REQUIRED    default false
  rollover_cap_tokens   integer       CONDITIONAL Required when rollover_enabled
= true

ENUM: TokenType
  INPUT_STANDARD        Standard prompt/input tokens
  OUTPUT_STANDARD       Standard completion/output tokens
  INPUT_CACHED          Input tokens served from cache (discounted)
  OUTPUT_EXTENDED_THINKING  Tokens generated during extended reasoning (premium)
  AGENT_ORCHESTRATION   Tokens consumed by agent planning and tool-call
orchestration
  EMBEDDING             Tokens used for embedding generation

OBJECT: AgentTaskCredit (pricing_model = AGENT_TASK)
  task_types            TaskType[]    REQUIRED    Which task types this plan
covers
  credit_per_task       integer       REQUIRED    Credits deducted per completed
task
  task_type_multipliers object        OPTIONAL    Override credit_per_task by
specific task type
  task_completion_definition string  REQUIRED    Human-readable definition of
what constitutes a completed task
  task_timeout_seconds  integer       REQUIRED    How long before an incomplete
task is considered failed (not billed)
  partial_task_billing  boolean       REQUIRED    default false; if true,
partially completed tasks are billed at a defined fraction

OBJECT: OutcomeDefinition
  id                    UUID          REQUIRED
  pricing_plan_id       UUID          REQUIRED
  outcome_description   string        REQUIRED    Plain-language description of
the outcome
  outcome_type          enum          REQUIRED    BINARY (did/did not occur) |
GRADED (quality score)
  verification_method   enum          REQUIRED    CUSTOMER_ACCEPTANCE |
AUTOMATED_SIGNAL | THIRD_PARTY_AUDIT | SAMPLING
  verification_signal   object        CONDITIONAL For AUTOMATED_SIGNAL: the event
schema and threshold
  verification_sla_days integer       REQUIRED    Days within which outcome
verification must be completed after delivery
  quality_tiers         QualityTier[] OPTIONAL    For GRADED outcomes: price per
outcome at each quality tier
  dispute_window_days   integer       REQUIRED    Days after billing during which
customer may dispute an outcome claim
```

**BUSINESS RULES**

**TOKEN-001:** A TokenProductConfig must specify at minimum input_token_cost and output_token_cost. All other token type costs default to the nearest applicable type cost if not specified.

| |
|---|
| **TOKEN-002:** input_token_cost must be less than or equal to output_token_cost. If an AI model charges more for input than output, this must be explicitly flagged as a configuration exception. |
| **TOKEN-003:** For HYBRID products containing a TOKEN component, the token component's included_tokens allocation is tracked separately per billing period and does not accumulate with the subscription floor component. |
| **AGENT-001:** AgentTaskCredit.task_completion_definition must be formally approved by the Product and Legal teams before the pricing plan can be activated. The definition becomes the contractual definition of a billable event. |
| **AGENT-002:** task_timeout_seconds must be set before activation. A task with no timeout creates unbounded billing liability — if the agent never completes, the customer is never charged, but the provider's infrastructure costs continue. |
| **OUTCOME-001:** OutcomeDefinition.verification_method = AUTOMATED_SIGNAL requires a verification_signal schema that is tested against a sample of 100+ historical outcomes before activation. |
| **OUTCOME-002:** A pricing plan with verification_method = CUSTOMER_ACCEPTANCE must include a dispute_window_days of at least 7 days to give customers a meaningful opportunity to dispute claimed outcomes. |
| **OUTCOME-003:** For GRADED outcomes, quality_tiers must cover all possible quality scores with no gaps. The lowest tier must have a price of $0 if outcomes below a minimum quality threshold are not billable. |

## Integration Checklist

□ Token product configurations follow the pattern used by major AI platforms — see Chapter 1.9 for AI platform implementation patterns

□ Agent task credits decouple the commercial unit (task) from the underlying resource (tokens) — enabling stable pricing as model efficiency improves

□ OutcomeDefinition is the contractual specification of a billable event — it must be formally approved before activation

□ Hybrid products are the most common AI commercial structure — a subscription floor provides revenue predictability; consumption overage captures value from heavy users

□ The TOKEN-003 rule prevents revenue recognition complexity when token components are part of hybrid products

CHAPTER 1.4

# Bundling: Design, Rules, Component Pricing, and Revenue Allocation

## Learning Objectives

• Design bundle structures with the appropriate component configuration for discount and capability bundles

• Select and document the correct allocation methodology (Relative Fair Value, Fixed Amount, or Residual) for each bundle

• Document standalone selling prices (SSPs) for all bundle components requiring relative fair value allocation

• Configure bundle validation rules that prevent allocation errors and downstream RevRec problems

• Specify the bundle lifecycle including component deprecation handling and bundle-level discount governance

## Process Flow

1. Bundle design: identify the products to be combined. Define whether the bundle is a discount bundle (components individually available, bundle offers a discount) or a capability bundle (bundle offers capabilities not available individually).
2. Bundle pricing determination: set the bundle price as either a specific Price object (the bundle price is distinct from component prices) or as a formula relative to component prices (e.g., sum of components minus 15%).
3. Component allocation methodology: determine how the bundle's total price is allocated across its components for revenue recognition. This allocation determines the timing of revenue recognition for each component. Methods: (a) Relative Fair Value — allocate proportionally to each component's standalone selling price (SSP); (b) Fixed Amount — assign fixed amounts to each component; (c) Residual — one component gets the residual after fixed amounts are allocated to others.
4. SSP documentation: for Relative Fair Value allocation, establish and document the SSP for each component. SSP is typically the component's standalone list price, or an observable transaction price if the list price does not reflect fair value.
5. Bundle discount interaction: define whether bundle-level discounts interact with component-level discounts. Best practice: bundle pricing is the final price; additional component-level discounts are not applied on top of bundle pricing without explicit deal desk approval.
6. Bundle activation: validate that all components are in ACTIVE status, the allocation methodology is configured, and SSPs are documented. Then activate the bundle.

## Data Model

```
OBJECT: Bundle (type = Product with product_type = BUNDLE)
  id                    UUID          REQUIRED    Same as Product.id
```

```
  component_products    BundleComponent[] REQUIRED  At least 2 components
  bundle_pricing_type   enum          REQUIRED    STANDALONE_PRICE |
FORMULA_BASED
  bundle_formula        object        CONDITIONAL Required when
bundle_pricing_type = FORMULA_BASED
  allocation_method     enum          REQUIRED    RELATIVE_FAIR_VALUE |
FIXED_AMOUNT | RESIDUAL
  ssps_documented       boolean       REQUIRED    Must be true before bundle can
be activated
  ssp_documentation_url string        CONDITIONAL Required when allocation_method
= RELATIVE_FAIR_VALUE

OBJECT: BundleComponent
  product_id            UUID          REQUIRED    FK → Product.id (must be
ACTIVE, is_standalone_sellable may be false for bundle-only components)
  quantity              integer       REQUIRED    Default quantity included in
bundle; min 1
  is_required           boolean       REQUIRED    Can the customer remove this
component from the bundle?
  standalone_selling_price integer    CONDITIONAL Required when
Bundle.allocation_method = RELATIVE_FAIR_VALUE; in cents
  fixed_allocation_amount integer     CONDITIONAL Required when
Bundle.allocation_method = FIXED_AMOUNT; in cents
  display_order         integer       REQUIRED    Ordering for quote and invoice
display
```

**BUSINESS RULES**

**BUNDLE-001:** A bundle must contain at least two BundleComponents. A single-component bundle is a product, not a bundle.

**BUNDLE-002:** Bundle price must be ≤ sum of component standalone selling prices. A bundle priced above the sum of components would deter rather than incentivize purchase — if intentional, it requires VP Finance approval and a documented rationale.

**BUNDLE-003:** Nested bundles (a Bundle component that is itself a Bundle) are limited to two levels. Three-level nesting is prohibited.

**BUNDLE-004:** When allocation_method = RELATIVE_FAIR_VALUE, all components must have documented SSPs before the bundle can be activated. SSP documentation must be dated and approved by the Finance Controller.

**BUNDLE-005:** A BundleComponent with is_required = false (optional component) must have an individual standalone price configured so the bundle can be re-priced if the customer removes the optional component.

**BUNDLE-006:** If a BundleComponent's Product is deprecated, the Bundle must be reviewed within 30 days. If no replacement component is configured within 30 days, the Bundle is automatically deprecated.

**BUNDLE-007:** Bundle-level discounts are applied to the bundle price, not to individual component prices. Applying a bundle discount at the component level changes the SSP ratios used for revenue allocation — this requires explicit RevRec team approval.

## Integration Checklist

□ Bundles are Products with product_type = BUNDLE — they follow the same lifecycle as other products

□ Component allocation methodology determines revenue recognition timing for each component — get it wrong and the RevRec team finds out at audit

□ SSP documentation is a compliance requirement, not an optional best practice — it is required for external audit defense

□ BUNDLE-006 (auto-deprecation when component is deprecated) prevents orphaned bundles with unavailable components

□ Bundle discounts are applied at the bundle level, not at the component level — mixing levels creates SSP ratio distortions

CHAPTER 1.5

# Discount Architecture: Types, Authority, Stacking Rules, and Accounting Treatment

## Learning Objectives

• Define the complete discount type taxonomy and map each type to its accounting treatment

• Design the discount authority matrix with appropriate authority levels for your organization's size and risk tolerance

• Configure stacking rules that prevent unintended discount combinations while allowing permitted stacking

• Specify the margin floor enforcement that prevents discounts from destroying product economics

• Build the discount governance audit trail that satisfies external audit requirements

## Process Flow

1. Discount type classification: every discount in the system has a type. The type determines the approval authority required, the accounting treatment, the stacking rules, and the expiry behavior.

2. Authority check: the discount type and amount are checked against the discount authority matrix. If the proposed discount exceeds the requesting user's authority, an approval request is automatically generated.

3. Stacking validation: if other discounts already exist for the same product/line, the system checks stacking rules. Some discount types are mutually exclusive (cannot stack), some are additive (amounts sum), and some are hierarchical (the higher discount wins, the lower is discarded).
4. Margin impact display: the billing or CPQ system displays the resulting gross margin after all discounts before the user can finalize the discount. Discounts taking margin below the product floor require escalated approval.
5. Discount record creation: the approved discount is recorded as a Discount object with a reference to the product, the approval chain, the effective date, and the expiry date.
6. Accounting treatment assignment: each discount type maps to an accounting treatment. Sales discounts reduce revenue. Volume rebates may be treated as a reduction of cost. Promotional credits may be contra-revenue or marketing expense depending on structure.

## Data Model

```
ENUM: DiscountType
  NEGOTIATED_DISCOUNT       Deal desk discount on a specific quote line
  VOLUME_DISCOUNT           Tiered discount based on cumulative purchase volume
  PROMOTIONAL_DISCOUNT      Time-limited discount for marketing/sales campaigns
  LOYALTY_DISCOUNT          Ongoing discount for long-tenure customers
  PARTNER_DISCOUNT          Discount applied through partner/reseller channel
  COMPETITIVE_DISCOUNT      Strategic discount to displace a competitor
  BETA_DISCOUNT             Discount for beta program participants
  GOODWILL_DISCOUNT         Customer service recovery discount
  MULTI_YEAR_DISCOUNT       Discount for committing to multi-year contracts
  PREPAYMENT_DISCOUNT       Discount for paying annual contract upfront
  BUNDLE_DISCOUNT           Implicit discount from bundle pricing
  REFERRAL_CREDIT           Credit earned through referral programme (see Credits
chapter)

OBJECT: DiscountPlan
  id                    UUID          REQUIRED
  discount_type         DiscountType  REQUIRED
  name                  string        REQUIRED
  discount_method       enum          REQUIRED    PERCENTAGE | FIXED_AMOUNT
  discount_value        integer       REQUIRED    Percentage in basis points
(1500 = 15%) or fixed cents
  applies_to            enum          REQUIRED    PRODUCT | PRODUCT_FAMILY |
LINE_ITEM | ENTIRE_ORDER
  target_id             UUID          CONDITIONAL Required when applies_to ≠
ENTIRE_ORDER
  effective_from        ISO 8601 date REQUIRED
  effective_to          ISO 8601 date OPTIONAL    null = perpetual (requires VP
Finance approval)
  stacking_behaviour    enum          REQUIRED    ADDITIVE | EXCLUSIVE |
HIERARCHICAL
  accounting_treatment  enum          REQUIRED    REVENUE_REDUCTION |
CONTRA_REVENUE | MARKETING_EXPENSE
  approval_status       enum          REQUIRED    PENDING | APPROVED | REJECTED |
EXPIRED
```

```
  approved_by           UUID          CONDITIONAL Required when approval_status =
APPROVED
  approval_justification string       OPTIONAL

OBJECT: DiscountAuthorityMatrix
  discount_type         DiscountType  REQUIRED
  max_percentage_ae     decimal       REQUIRED    Maximum % an Account Executive
can approve autonomously (e.g., 0.10 = 10%)
  max_percentage_manager decimal      REQUIRED    Maximum % a Sales Manager can
approve
  max_percentage_vp     decimal       REQUIRED    Maximum % a VP Sales can
approve
  max_percentage_cfo    decimal       REQUIRED    Maximum % a CFO can approve
  above_cfo_requires    string        REQUIRED    Description of approval
required above CFO level
```

| BUSINESS RULES |
|---|
| **DISC-001:** All discounts must have an effective_from and an explicit effective_to date OR a perpetual flag with VP Finance approval documented. Open-ended discounts without documentation are non-compliant. |
| **DISC-002:** Promotional and competitive discounts are mutually exclusive (stacking_behaviour = EXCLUSIVE). If both apply to the same line, the larger discount is applied and the smaller is discarded with an audit log entry. |
| **DISC-003:** A NEGOTIATED_DISCOUNT and a VOLUME_DISCOUNT may stack (ADDITIVE) but the combined discount cannot exceed the authority level of the approver who approved the negotiated discount. |
| **DISC-004:** Any discount taking gross margin below the product-level margin floor requires CFO approval regardless of the requesting user's standard discount authority. |
| **DISC-005:** GOODWILL_DISCOUNT has no standard authority level — it always requires Manager approval minimum and must have a customer support ticket reference in the approval justification. |
| **DISC-006:** Discounts applied to bundle prices are recorded at the bundle level, not at the component level. Applying a discount at the component level within a bundle changes SSP ratios and requires RevRec team sign-off. |
| **DISC-007:** Discount records are immutable once approved. Corrections require creating a new discount with the corrected terms and revoking the original with an audit log entry. |

## Integration Checklist

□ Twelve discount types cover all commercial scenarios — each type has a distinct accounting treatment that must be configured before the discount can be issued

□ The authority matrix is the primary control preventing unauthorized price reductions — it must be approved by the CFO before the discount system goes live

□ Stacking rules are configured at the DiscountPlan level — EXCLUSIVE stacking is the safest default for promotional discounts

□ DISC-004 (margin floor enforcement) is the most commercially important business rule in this chapter — without it, discounts can make products unprofitable

□ Perpetual discounts (no effective_to date) require VP Finance approval under DISC-001 — this prevents the common problem of temporary discounts becoming permanent

CHAPTER 1.6

# Credits and Promotional Economics: Account Credits, Vouchers, Referral Credits

## Learning Objectives

• Distinguish credits from discounts and understand the accounting treatment implications of each credit type

• Design the credit issuance workflow for each credit type including authorization requirements

• Configure the credit application priority order and the rules governing application to specific invoices

• Specify expiry handling and the accounting treatment for expired credits by type

• Design the customer-facing credit visibility features required in the customer portal

## Process Flow

Credits and promotional economics are distinct from discounts: where a discount reduces the price of a product, a credit reduces the amount owed on an invoice or account balance after billing has occurred. Credits are balance-sheet items (they create a liability or contra-revenue entry) where discounts are income-statement items (they reduce revenue directly at point of sale).

1. Credit type selection: determine the credit type from the taxonomy. The type determines accounting treatment, issuance authorization, application priority, and expiry behavior.
2. Credit issuance: credits are issued against a specific customer account. The issuance event creates a Credit object with a balance, an issuance reason, and an expiry date.
3. Application to invoices: credits are applied during billing run or manually post-invoice. The application order is: SLA credits first (they are contractual obligations), then account credits by issuance date (oldest first), then vouchers.

4. Expiry handling: credits that reach their expiry date without being applied are expired. Expired credits are written off. The accounting entry for expiry depends on the original credit type (contra-revenue reversal for SLA credits; income recognition for expired loyalty credits).
5. Credit balance visibility: the customer must be able to see their current credit balance, credit history, and upcoming expiry dates in the customer portal.

## Data Model

```
ENUM: CreditType
  SLA_CREDIT              Issued due to SLA breach; contractual obligation
  GOODWILL_CREDIT         Issued for customer service recovery; management
approval required
  PROMOTIONAL_CREDIT      Issued as part of marketing campaign or promotion
  REFERRAL_CREDIT         Earned through referral programme
  BILLING_CORRECTION      Issued to correct a billing error
  LOYALTY_CREDIT          Earned through tenure or spend milestones
  VOUCHER                 Redemption code with a defined credit value

OBJECT: AccountCredit
  id                    UUID          REQUIRED
  account_id            UUID          REQUIRED    FK → Account.id
  credit_type           CreditType    REQUIRED
  amount                integer       REQUIRED    Credit amount in cents; must be
positive
  currency              ISO 4217      REQUIRED
  issued_at             ISO 8601 UTC  SYSTEM
  issued_by             UUID          REQUIRED    User ID of issuer
  issuance_reason       string        REQUIRED    Human-readable reason
  reference_id          UUID          OPTIONAL    FK → SLABreach.id or
SupportTicket.id
  effective_from        ISO 8601 date REQUIRED    Date credit becomes usable
  expires_at            ISO 8601 date REQUIRED    Credits must have an expiry
date
  status                enum          REQUIRED    ACTIVE | PARTIALLY_APPLIED |
FULLY_APPLIED | EXPIRED | REVOKED
  remaining_balance     integer       SYSTEM      Maintained by system; = amount
minus sum of applications
  accounting_treatment  enum          REQUIRED    CONTRA_REVENUE | LIABILITY |
DEFERRED_REVENUE
```

**BUSINESS RULES**

**CREDIT-001:** All credits must have an expires_at date. Perpetual credits are prohibited. The standard minimum credit validity is 90 days; exceptions require VP Finance approval.

**CREDIT-002:** SLA_CREDIT is the only credit type that can be issued automatically by the system (triggered by SLA breach detection). All other credit types require human issuance with an approval workflow.

**CREDIT-003:** Credits are applied to invoices in the following priority order: (1) SLA credits — applied to the invoice for the period in which the breach occurred, (2) BILLING_CORRECTION credits — applied to the corrected invoice, (3) all other credits by oldest first.

| |
|---|
| **CREDIT-004:** A VOUCHER cannot be applied to an invoice that has already been paid. Vouchers can only reduce the amount owed, not generate a cash refund. |
| **CREDIT-005:** Credit issuance of GOODWILL_CREDIT requires a customer support ticket reference. Credits issued without a support reference are non-compliant and will fail the audit review. |
| **CREDIT-006:** The accounting treatment of expired credits: SLA_CREDIT expiry reverses the original SLA credit liability entry (revenue is reinstated). LOYALTY_CREDIT expiry is recognized as other income. PROMOTIONAL_CREDIT expiry reverses the contra-revenue entry. |

## Integration Checklist

□ Credits are balance-sheet items (create a liability or contra-revenue entry); discounts are income-statement items (reduce revenue at point of sale)

□ SLA_CREDIT is the only credit type that can be issued automatically — all others require human authorization

□ Application priority order (SLA credits first, then billing corrections, then others oldest-first) prevents customers from gaming the credit application sequence

□ CREDIT-001 (mandatory expiry date) is critical — perpetual credits create indefinite accounting liabilities

□ Expired credit accounting treatment differs by type — SLA credits reverse the liability; promotional credits reverse contra-revenue — the RevRec system must handle each correctly

CHAPTER 1.7

# Pricing Governance: Approval Workflows, Change Management, and Audit Trail

## Learning Objectives

• Design the pricing governance workflow that ensures all catalog changes are reviewed, approved, and documented

• Configure the change request routing rules that direct requests to the appropriate approval authorities

• Specify the customer notification requirements for different categories of pricing changes

• Build the effective date management logic that prevents changes from taking effect prematurely

• Define the audit trail requirements that satisfy external audit standards for pricing governance

## Process Flow

Pricing governance is the operational architecture that prevents unauthorized price changes, ensures adequate review before price changes affect customers, and maintains the audit trail required for external scrutiny.

1. Change initiation: any user with PRICING_PROPOSE access can initiate a pricing change request. The request captures the proposed change, the justification, the affected products/plans, the proposed effective date, and the estimated revenue impact.
2. Routing: the system routes the request based on the change type and estimated revenue impact. Price increases above 10% on any active product require VP Finance in the approval chain. Any change affecting more than $1M in ARR requires CFO approval.
3. RevRec review: any change to a pricing structure (new pricing model, change to existing model, change to allocation methodology) automatically routes to the RevRec team for ASC 606 impact assessment before final approval.
4. Customer impact notification: if approved, the system generates a list of affected customers who are currently on the changing pricing plan. The catalog owner reviews the list and approves customer notifications before the change takes effect.
5. Effective date enforcement: the approved change does not apply until the effective date. On the effective date, the new pricing plan becomes active. Quotes generated before the effective date are not retroactively updated; quotes generated after use the new pricing.
6. Audit trail: every step — proposal, routing, review, approval/rejection, notification, effective date activation — is recorded in an immutable audit log.

## Data Model

```
OBJECT: PricingChangeRequest
  id                    UUID          REQUIRED
  request_type          enum          REQUIRED    NEW_PRODUCT | PRICE_CHANGE |
MODEL_CHANGE | DEPRECATION | BUNDLE_CHANGE
  product_id            UUID          REQUIRED
  pricing_plan_id       UUID          OPTIONAL    Specific plan if change is
plan-level
  proposed_change       object        REQUIRED    Before/after diff of the
proposed change
  business_justification string       REQUIRED    min 50 characters
  proposed_effective_date ISO 8601 date REQUIRED  Must be in future unless
PRICING_ADMIN with override
  estimated_revenue_impact_cents integer REQUIRED Positive = revenue increase;
negative = decrease
  status                enum          REQUIRED    DRAFT | SUBMITTED | IN_REVIEW |
REVREC_REVIEW | APPROVED | REJECTED | CANCELLED | APPLIED
  submitted_by          UUID          REQUIRED
  submitted_at          ISO 8601 UTC  SYSTEM
  approvers             Approval[]    SYSTEM      Populated by routing rules
  revrec_assessment_id  UUID          OPTIONAL    FK → RevRecAssessment.id
```

```
(populated when routed to RevRec)
  affected_customers    UUID[]        SYSTEM      Populated on approval; reviewed
by catalog owner before effective date
  customer_notification_approved_by UUID CONDITIONAL Required before change takes
effect if affected_customers > 0

OBJECT: Approval
  approver_user_id      UUID          REQUIRED
  required_role         string        REQUIRED    Role required to fulfill this
approval
  status                enum          REQUIRED    PENDING | APPROVED | REJECTED |
DELEGATED
  decided_at            ISO 8601 UTC  OPTIONAL
  decision_comment      string        OPTIONAL
```

**BUSINESS RULES**

**GOV-001:** All pricing changes require a PricingChangeRequest. Direct modification of PricingPlan or Price objects without a corresponding approved PricingChangeRequest is a control violation and will be flagged in audit.

**GOV-002:** Pricing changes cannot have a proposed_effective_date more than 12 months in the future. Long-dated price changes require re-approval if market conditions change.

**GOV-003:** Any PricingChangeRequest with estimated_revenue_impact_cents > 1,000,000,000 (i.e., > $10M ARR impact) requires Board notification (not approval, but notification) before taking effect.

**GOV-004:** Retroactive pricing changes (effective date in the past) require CFO approval and a RevRec assessment. They are generally prohibited except for billing corrections or contractual obligations.

**GOV-005:** When a PricingChangeRequest is rejected, the rejection reason must be documented in the decision_comment field. Rejections without documented reasons are non-compliant.

**GOV-006:** Customer notification for price increases must be sent at least 30 days before the effective date for customers with annual contracts, and at least 14 days for customers with monthly contracts, unless the customer's contract specifies a different notification period.

## Integration Checklist

□ All pricing changes must have an approved PricingChangeRequest — direct object modification is a control violation

□ RevRec review is mandatory for any change to pricing structure, not just pricing amounts

□ Customer notification lead time is contractually defined — the system enforces the minimum based on contract type

□ GOV-004 (retroactive changes require CFO approval) prevents retroactive corrections that create prior-period revenue adjustments

□ The change impact analysis agent (Chapter 1.9) automates the affected customer identification that previously required manual analysis

CHAPTER 1.8

# Catalog Versioning: Version Control, Effective Dating, and Customer Migration

## Learning Objectives

• Design the catalog versioning architecture that allows the catalog to evolve without breaking existing customer relationships

• Configure customer migration strategies for new catalog versions including auto-migration, opt-in, and grandfather arrangements

• Specify the version diff format that enables customer communication and internal review of changes

• Define the legacy version management lifecycle including the conditions for READ_ONLY and ARCHIVED status

• Build the audit trail for catalog version changes that documents who activated each version and when

## Process Flow

The catalog versioning architecture enables the product catalog to evolve over time — new products added, pricing updated, models changed — without breaking existing customer relationships or creating ambiguity about which version of the catalog governs a specific invoice.

1. Version creation: a new catalog version is created when a set of catalog changes needs to be bundled together for a coordinated release. Version creation is separate from individual change requests — a version can contain multiple approved changes.

2. Draft and testing: the new version is in DRAFT status during development. The CPQ system and billing engine can be configured to run in test mode against the DRAFT version to validate that quote generation and billing produce expected results.

3. Activation scheduling: the catalog owner schedules the version for activation on a specific date. The activation date must satisfy the customer notification requirements for any price-increasing changes in the version.

4. Customer assignment: each active customer account has a catalog_version_id that determines which version of the catalog governs their billing. By default, customers are migrated to the new version on its effective date. Exceptions: grandfathered customers remain on the previous version; customers with contracts locking specific pricing remain until contract renewal.

5. Legacy version management: previous catalog versions are maintained in READ_ONLY status for as long as active contracts reference them. A version can only be ARCHIVED when no active contracts reference it.
6. Version diff: the system provides a machine-readable diff between any two catalog versions, showing all added, changed, and removed objects. This diff is used for customer notification and internal review.

## Data Model

```
OBJECT: CatalogVersion
  id                    UUID          REQUIRED
  version_number        string        REQUIRED    Semantic version: YYYY.MM.PATCH
(e.g., 2025.Q3.2)
  status                enum          REQUIRED    DRAFT | SCHEDULED | ACTIVE |
READ_ONLY | ARCHIVED
  effective_date        ISO 8601 date REQUIRED
  description           string        REQUIRED    What changed in this version
  created_by            UUID          REQUIRED
  approved_by           UUID          CONDITIONAL Required before SCHEDULED
status
  activated_at          ISO 8601 UTC  SYSTEM
  included_changes      UUID[]        REQUIRED    Array of
PricingChangeRequest.id included in this version
  migration_strategy    enum          REQUIRED    AUTO_MIGRATE | REQUIRE_OPT_IN |
GRANDFATHER_ELIGIBLE

OBJECT: CustomerCatalogAssignment
  customer_id           UUID          REQUIRED
  catalog_version_id    UUID          REQUIRED
  assigned_at           ISO 8601 UTC  SYSTEM
  assigned_by           UUID          REQUIRED    SYSTEM for auto-migration; user
ID for manual assignment
  assignment_reason     enum          REQUIRED    AUTO_MIGRATION | CONTRACT_LOCK
| GRANDFATHERED | MANUAL_OVERRIDE
  valid_through         ISO 8601 date OPTIONAL    For CONTRACT_LOCK and
GRANDFATHERED assignments
```

| BUSINESS RULES |
| --- |
| **VERSION-001:** A new CatalogVersion must include at least one approved PricingChangeRequest. Empty versions are not permitted. |
| **VERSION-002:** A CatalogVersion in DRAFT status does not affect any live customer billing or quoting. The CPQ and billing systems use the currently ACTIVE version unless explicitly configured to test against a DRAFT version. |
| **VERSION-003:** The gap between version scheduling and activation (scheduled_date minus today) must be sufficient to satisfy all customer notification requirements. The system enforces this — if any included change requires 30-day notice, the activation date cannot be less than 30 days in the future. |

| |
|---|
| **VERSION-004:** Customers with CONTRACT_LOCK assignments are not migrated to new versions automatically. Their assignment remains tied to the version valid at contract signing until the contract renewal date. |
| **VERSION-005:** GRANDFATHERED customers must have an explicit valid_through date. Grandfather arrangements without an end date require VP Finance approval. |
| **VERSION-006:** A CatalogVersion can only be ARCHIVED when all CustomerCatalogAssignments referencing it have migrated to newer versions. Archiving a version with active assignments is prevented by the system. |

## Integration Checklist

□ Catalog versioning separates 'what changed' (PricingChangeRequests) from 'when it takes effect for customers' (CatalogVersions)

□ CustomerCatalogAssignment is the object that tracks which version governs each customer — it is the single source of truth for billing version selection

□ CONTRACT_LOCK assignments are never auto-migrated — they require explicit action at renewal

□ VERSION-006 (version archival only when no active assignments reference it) prevents orphaned billing configurations

□ The version diff is machine-readable — it feeds directly into the customer notification workflow and the change impact analysis

CHAPTER 1.9

# AI Agents for Product Catalog Management

*The four AI agents that provide meaningful productivity improvement in product catalog management — with full specifications for each*

Four AI agents provide meaningful productivity improvement in product catalog operations. Each is specified here at the level required to build: the agent's function, the data it needs access to, the tools it uses, the guardrails that limit its authority, the outputs it produces, and the escalation conditions that require human involvement. The governing principle for all catalog agents: agents may read, analyse, and recommend, but they may not modify catalog objects without human approval. The catalog is a governance-critical data structure — unauthorized changes have downstream billing consequences.

## Agent 1: Catalog Health Monitoring Agent

| Catalog Health Monitoring Agent — Specification | |
|---|---|
| **Attribute** | **Specification** |
| Function | Continuously monitors the product catalog for data quality issues, coverage gaps, and governance violations |
| Data access | Read access to all catalog objects (Products, PricingPlans, Prices, Bundles, CatalogVersions, CustomerCatalogAssignments) |
| Tools | catalog_health_scan, missing_field_detector, rule_violation_scanner, coverage_gap_analyser, expiry_tracker |
| Guardrails | READ-ONLY — cannot modify any catalog object · Cannot approve or reject PricingChangeRequests · Cannot reassign customer catalog versions |
| Immediate alerts (real-time) | PricingPlan with no active Price object for >24 hours · Product with status ACTIVE and no active PricingPlan · Bundle with deprecated component not reviewed within 30-day window |
| Daily alerts | PricingPlans expiring within 60 days with no replacement · Customers on catalog version >2 versions behind current · Products with missing SSP documentation required for bundles |
| Escalation | Alerts route to Pricing Operations team via notification framework. Unacknowledged critical alerts escalate to Pricing Manager after 4 business hours. |
| Outputs | Daily catalog health dashboard report. Real-time alerts on critical issues. Weekly trend analysis on catalog complexity (product count, version sprawl, average active plans per product). |

## Agent 2: Pricing Conflict Detection Agent

| Pricing Conflict Detection Agent — Specification | |
|---|---|
| **Attribute** | **Specification** |
| Function | Detects logical conflicts and inconsistencies in pricing configurations before they manifest as billing errors or customer disputes |
| Data access | Read access to all pricing objects; read access to recent billing exceptions (to identify patterns) |
| Tools | tier_gap_detector, rule_validation_engine, cross_product_conflict_scanner, billing_impact_estimator |
| Checks performed | TIERED pricing tiers have no gaps or overlaps · HYBRID product |

| | |
|---|---|
| | component allocation sums to 100% · TOKEN pricing with included_tokens has overage_handling configured · OUTCOME pricing has verification_method specified · Bundle allocation requires SSP documentation present · Discount plans with EXCLUSIVE stacking have no overlap in effective date ranges for same product |
| Guardrails | READ-ONLY · Cannot modify any pricing configuration · Conflict alerts require human review before any action is taken |
| Outputs | Conflict report attached to every PricingChangeRequest before routing to approvers. Real-time conflict detection when new pricing objects are saved in DRAFT status. |

## Agent 3: Pricing Approval Routing Agent

| Pricing Approval Routing Agent — Specification | |
|---|---|
| **Attribute** | **Specification** |
| Function | Determines the correct approval chain for each PricingChangeRequest based on the authority matrix, estimated revenue impact, and change type |
| Data access | Read access to DiscountAuthorityMatrix, user roles and reporting lines, PricingChangeRequest details, active contract data for revenue impact calculation |
| Tools | authority_matrix_resolver, revenue_impact_calculator, approver_availability_checker, delegation_resolver, sla_monitor |
| Guardrails | Can route requests but cannot approve them · Cannot modify authority matrix rules · Cannot bypass required approval levels · Delegation must be pre-configured by an active approver — agent cannot create ad-hoc delegations |
| Escalation triggers | Primary approver has not responded within defined SLA · Primary approver is on leave with no delegation configured · Approval chain cannot be completed due to organizational changes |
| Outputs | Routing decision with justification attached to each PricingChangeRequest. SLA countdown timer visible to all parties in approval chain. Escalation notification when SLA is breached. |

## Agent 4: Change Impact Analysis Agent

| Change Impact Analysis Agent — Specification | |
|---|---|
| **Attribute** | **Specification** |
| Function | Analyses the commercial impact of a proposed catalog change before it is |

| | |
|---|---|
| | approved — identifying affected customers, estimating revenue impact, and flagging RevRec implications |
| Data access | Read access to catalog objects, active contracts, billing history, CustomerCatalogAssignments, RevRec recognition schedules |
| Tools | affected_customer_scanner, revenue_impact_modeller, revrec_flag_detector, customer_notification_list_generator |
| Guardrails | READ-ONLY across all systems · Cannot modify contracts, pricing, or RevRec schedules · Revenue impact estimates are flagged as estimates — not used for financial reporting without human validation |
| Required outputs | List of customers affected by the change · Estimated ARR impact (increase/decrease) with confidence interval · RevRec flags: does this change create a contract modification under ASC 606? · Notification list: customers requiring advance notice with notification dates |
| Trigger | Automatically triggered when a PricingChangeRequest moves from DRAFT to SUBMITTED status. Output is attached to the request before it reaches the first approver. |

**Chapter 1.9 — Key Takeaways**

› Four agents cover the four highest-value automation opportunities in catalog management: health monitoring, conflict detection, approval routing, and change impact analysis

› All four agents are READ-ONLY — they analyse and recommend; they do not modify catalog objects without human approval

› The Conflict Detection Agent is triggered by the creation or modification of any pricing object in DRAFT status — it prevents conflicts from reaching production

› The Change Impact Agent fires automatically when a PricingChangeRequest is submitted — first approver always sees the impact analysis

› Agent actions are logged to the audit trail with the agent ID as actor — audit trail attribution is not lost when agents perform automated analysis

APPENDIX A

# AI Agent Reference

*Consolidated specification for all four AI agents in ROS-01 — for integration teams building agent infrastructure*

This appendix provides the consolidated reference for all AI agents specified in ROS-01. Each agent entry below follows the standard RevenueOS agent specification format: name, module, function, data access requirements, tools, guardrails, escalation logic, and outputs. General governance rules applicable to all agents in this module: 1. All agents in the product catalog module are READ-ONLY — no agent may modify any catalog object without human approval. 2. Agent actions are logged to the audit trail with the agent ID as the actor, not the user who deployed the agent. 3. All agent alerts must include the specific rule or condition that triggered the alert, the affected object IDs, and a recommended human action. 4. Agents must not make pricing recommendations that are based on competitor-specific information unless that information has been legally obtained and documented.

> **GOVERNANCE PRINCIPLE**
>
> **All catalog agents are READ-ONLY — agents may not modify catalog objects without human approval**
>
> The product catalog is governance-critical. Unauthorized changes create downstream billing consequences that may not surface for weeks. The four agents in this module provide intelligence and automation at the analysis and routing layer — the modification layer remains under human control.

**ROS-01 AI Agents — Quick Reference**

| Agent | Trigger | Primary Output | Authority | Escalation |
|---|---|---|---|---|
| Catalog Health Monitor | Continuous (scheduled) | Daily health report + real-time critical alerts | READ-ONLY | Unacknowledged alerts → Pricing Manager after 4 hours |
| Pricing Conflict Detection | On pricing object save (DRAFT status) | Conflict report attached to change request | READ-ONLY | Conflicts blocking activation → Pricing Manager |
| Approval Routing | On PricingChangeRequest submission | Routing decision with approval chain | READ-ONLY; routes only | SLA breach → escalate to next authority level |
| Change Impact Analysis | On PricingChangeRequest | Impact report: affected | READ-ONLY | Incomplete analysis (missing |

| | | | | |
|---|---|---|---|---|
| | DRAFT → SUBMITTED | customers, ARR impact, RevRec flags, notification list | | data) → notify Pricing Ops |

CLOSING

# The Foundation Is Set

*Everything downstream builds on what you defined here.*

The product catalog is the commercial foundation that every other module in the RevenueOS stack depends on. Every chapter that follows — contract lifecycle management, order management, billing, revenue recognition — assumes that the products being referenced are well-defined, the pricing is correctly configured, the bundles are properly allocated, and the governance records are complete.

The investment in getting the catalog right is disproportionately valuable. A billing engine built on a clean, consistent, version-controlled catalog produces accurate invoices with minimal engineering effort. A billing engine fighting against a catalog of inconsistent SKUs, missing SSPs, and undocumented special cases consumes engineering capacity in proportion to the catalog debt.

The nine chapters of this book give a PM, a finance engineer, and an integration team the specification to build and govern a product catalog that ages well — one that can be changed safely, audited cleanly, and extended without breaking what already works.

The rest of the series builds on this foundation. Module 2 (CLM) captures the contracts that reference these catalog items. Module 4 (Deal Desk) quotes from them. Module 9 (Billing) invoices against them. Module 19 (Revenue Accounting) recognizes revenue based on them. Get the catalog right first.

> ***"Get the catalog right. Everything downstream depends on it."***

---