BOOK TWO-A · THE AI ECONOMY MONETIZATION SERIES

The Data and Pricing Foundation

Monetization objects, the golden thread, and the five pricing layers — the bedrock of AI revenue architecture

Every billing failure has a data root. Every revenue leak has a schema it escaped through. Fix the model first.

Get the data model right before you build anything else. Pricing strategy is architecture.

Audience: Product managers, architects, pricing strategists, RevOps leads

PREFACE

Monetization Is an Architecture Problem

Why the data model must come before the process, and the process before the system.

This book begins with a premise that most software companies resist: monetization is an architecture problem.

Not a pricing problem. Not a process problem. Not a billing software problem. An architecture problem — in the precise engineering sense that the structural decisions made early constrain everything that follows, that mistakes made at the foundation are expensive to correct at the surface, and that the right approach is to design deliberately before building anything.

The architects who read this book will understand this immediately. The pricing strategists and product managers will need a moment. But the argument is the same one that governs software architecture: if your data model is wrong, your system will not work correctly, no matter how well you write the code that runs on top of it. If your monetization data model is wrong, your billing system will not work correctly, no matter how well the billing software is configured, no matter how many people you hire to manage it manually.

The monetization data model is the set of objects — their definitions, their schemas, their relationships, their lifecycle states — that represents every commercially significant entity in your AI business. Products. Prices. Entitlements. Meters. Events. Invoices. Contracts. Credits. Allocations. Token budgets. Agent tasks. Outcomes. Assets. Get these thirteen objects right, with precision, before you design a single billing process or evaluate a single billing platform. Everything else is downstream of this decision.

This is Part One of the book. Part Two covers the pricing of each of the five AI economy layers — compute, model, token, agent, outcome — at the level of implementation detail required to actually build a pricing architecture, not just choose one. Part Three covers offer design and catalog: how you translate a set of well-defined products and prices into a commercial catalog that can support CPQ, contracting, billing, and revenue recognition.

The book is written for practitioners: the people who will actually implement the systems described. It assumes familiarity with software product management and the basics of revenue operations. It does not assume expertise in billing systems or pricing theory — those are what the book is designed to build.

PART ONE

The Data Foundation

Thirteen canonical objects. The complete vocabulary of AI commerce.

CHAPTER ONE

Monetization Objects: The Universal Data Model

Thirteen objects. Precise definitions. Explicit relationships. The architecture that every other decision depends on.

The data model is the most important decision in monetization architecture. It is also the decision that is most frequently not made at all — replaced instead by the implicit data model of whatever billing platform the company happened to buy, or whatever CRM fields the sales team happened to fill in, or whatever spreadsheet the billing ops team built to track what the billing platform could not handle. Every one of those implicit data models is wrong in some specific way. The billing platform's data model was designed for SaaS subscriptions, not AI consumption billing. The CRM's data model was designed for opportunity tracking, not entitlement governance. The spreadsheet's data model was designed to solve last month's immediate problem, not to support the commercial architecture the company will need in three years. The explicit monetization data model — the one this part of the book is about — is designed deliberately, with full knowledge of the commercial requirements it must satisfy, before any system is selected to implement it.

Why the Data Model Must Come First

The question practitioners ask most frequently when this approach is introduced is: why does the data model need to come first? Why not start with the process — the billing workflow, the quote-to-cash sequence — and let the data model emerge from the process design?

The answer is that data models are much harder to change than processes. A process can be adjusted without touching the underlying data: add an approval step, remove a review

gate, change the sequence of tasks. But a change to the data model — adding a field, changing the cardinality of a relationship, splitting one object into two — requires changes to every process and every system that depends on that object. In a billing context, that means changes to the quote system, the contract system, the entitlement system, the metering system, the billing engine, the revenue recognition module, and the financial reporting system. All simultaneously.

The cost of a wrong data model is measured in months of engineering time, in billing errors that accumulate while the model is being corrected, and in revenue that leaks through the gaps created by the inconsistency between what the data model says and what the business actually requires.

Three specific patterns reveal an incorrect monetization data model. The first is the spreadsheet exception: a spreadsheet that exists alongside the billing system to handle cases the billing system cannot. Every spreadsheet exception is a data model failure — a commercial scenario that the data model cannot represent, so humans are routing around it manually. The second is the billing dispute that cannot be resolved: a customer challenges a charge, and the vendor cannot trace the charge back to its source events because the event data is not connected to the invoice data in a queryable way. The third is the quarter-close crisis: a month-end reconciliation that requires heroic manual effort because the data in multiple systems is inconsistent in ways that accumulate across the period.

If your organization has any of these three patterns, you have a data model problem. The right response is not to add more manual process around the problem. It is to fix the data model.

> ***"A change to the data model requires changes to every system that depends on it. A change to a process requires***

> ***nothing more than a policy update. Fix the model first — before the process, before the system."***

⚠ The Three Warning Signs of a Broken Data Model

If your organization has any of these three patterns, you have a data model problem — not a process problem and not a software problem. (1) A spreadsheet that exists alongside the billing system to handle cases the billing system cannot. Every spreadsheet exception is a data model failure. (2) A billing dispute that cannot be resolved by tracing the charge to its source events. Unresolvable disputes mean the event-to-invoice traceability chain is broken. (3) A quarter-close crisis requiring heroic manual effort to reconcile inconsistent data across multiple systems. Manual close effort is the accumulated cost of data model debt.

Introducing the Thirteen Objects

The thirteen monetization objects are the complete vocabulary needed to describe any AI commercial scenario. They are not a software architecture — they are a business architecture. The definitions that follow are the canonical definitions: precise enough to implement in any system, but not tied to any specific system's implementation.

Before reading the definitions, note their structure. Every object has a precise definition of what it represents. Every object has a lifecycle — a defined set of states it can be in and a defined set of transitions between those states. Every object has a defined set of relationships to other objects. And every object has a defined purpose in the commercial lifecycle — what decisions it governs, what events it generates, what downstream objects it affects.

This structure is not bureaucratic overhead. It is what makes the objects interoperable — what allows a Product defined in the product catalog to be referenced by a Price, governed by an Entitlement, measured by a Meter, recorded in an Event, and billed through an Invoice, with each system confident that it is talking about the same thing.

The Thirteen Monetization Objects — Master Reference			
Object	**Commercial role**	**Primary relationships**	**Key lifecycle states**
Product	What is being sold — the commercial wrapper for an AI capability	→ Price (defines how charged) · → Entitlement (defines what customer can consume)	draft → active → deprecated → archived
Price	How the product is charged — pricing logic and rate structure	→ Product (what it prices) · → Entitlement (pricing terms in force)	draft → active → deprecated
Entitlement	What the customer has the right to consume right now	→ Contract (authority) · → Meter (measurement config) · → Token Budget (hierarchy)	pending_activation → active → suspended → expired
Meter	How consumption is measured — the measurement instrument	→ Product (what it measures) · → Event (raw data it governs)	active · deprecated
Event	Atomic record of a single commercially significant occurrence	→ Entitlement (context) · → Agent Task (parent, if applicable)	created (immutable thereafter)
Invoice	Formal payment request derived from aggregated events	→ Contract · → Events (via line items and event_summary_uri)	draft → issued → disputed → paid
Contract	Binding commercial agreement — source of all entitlement terms	→ Entitlement (authorizes) · → Allocation (rev rec) · → Invoice (governs)	draft → active → amended → expired
Credit	Reduction in amount owed — always requires authorization	→ Invoice (reduces) · → Approval event (authorizes)	pending → approved → applied
Allocation	Revenue recognition distribution across performance obligations	→ Contract (what it allocates) · → Revenue recognition entries	draft → active → amended
Token Budget	Hierarchical financial control for AI spending	→ Entitlements in scope · → Approval workflow (for increases)	active · warning · exhausted · reset

Agent Task	Single execution of an autonomous AI workflow	→ Events (children) · → Entitlement (governed by) · → Workflow (definition)	queued → running → completed / failed
Outcome	Verified business result that triggers outcome-based billing	→ Outcome definition · → Billing event (triggered by verification)	pending_verification → verified → rejected
Asset	Per-customer inventory of active AI capabilities	→ Product · → Entitlement · → Contract	active → pending_renewal → expired

CHAPTER TWO

Object Deep Dives — Products, Prices, and Entitlements

The three objects that govern commercial intent. Get these wrong and everything downstream is wrong.

Object 1 — Product

The Product object represents an AI capability packaged for commercial sale. It is the top of the commercial hierarchy — every pricing decision, entitlement, metering configuration, and invoice derives ultimately from a Product.

A Product is not the AI model. It is not the infrastructure. It is the commercial representation of a capability that can be sold. The same AI model may be sold as multiple Products — different usage tiers, different deployment models, different geographic restrictions — each with its own commercial terms.

What makes a Product definition precise is the specificity of its type declaration and its AI layer designation. The type declaration determines which pricing models are valid for this product and which entitlement structures are required. A token_bundle product is priced per token and governed by a token budget entitlement. An agent_workflow product is priced per task and governed by a task limit entitlement. Confusing the types

— creating a product as a subscription type but trying to apply per-task pricing to it — is a data model error that will eventually surface as a billing error.

The AI layer designation — compute, model, token, agent, or outcome — is equally important. It determines the natural pricing logic for this product, which influences the Meter configuration, the Entitlement structure, and the revenue recognition treatment. A product designated at the outcome layer is subject to variable consideration treatment under ASC 606 in a way that a product designated at the compute layer is not.

Product versioning deserves particular attention in AI contexts. Foundation models improve continuously. The AI product sold in January 2025 uses a different model than the same product in January 2026. Whether this represents a new Product (which would require new Entitlements and potentially new Prices) or a version update to an existing Product (which preserves existing commercial relationships) is a data governance decision that affects every downstream object. Most companies handle this inconsistently because they have not established a versioning policy. Establishing one early — with clear criteria for when a capability change is a new product versus a version update — prevents significant commercial complexity later.

Product Object — Complete Field Reference

Field	Type	Description / Values
`id`	UUID v4 · required	Globally unique. Generated at creation. Never reused or reassigned.
`name`	string · required	Human-readable display name. Max 200 chars. Used in invoices and customer portal.
`type`	enum · required	token_bundle | agent_workflow | outcome_service | model_access | compute_reservation | platform_subscription | composite
`ai_layer`	enum · required	compute | model | token | agent | outcome — determines natural pricing logic and rev rec treatment
`status`	enum · required	draft | active | deprecated | archived

`version`	semver · required	Incremented on any substantive capability change. Drives versioning policy decisions.
`description`	string · optional	Full capability description. Used by catalog search and agent discovery.
`capability_description`	string · recommended	Natural language description optimized for semantic search by AI agents.
`parent_product_id`	UUID · optional	For composite products. References the parent Product this is a component of.
`model_ids`	UUID[] · optional	Foundation models this product uses. Affects cost attribution and FinOps reporting.
`pricing_reference`	UUID · optional	Convenience reference to the primary Price object. Does not replace the one-to-many Price relationship.
`created_at`	ISO 8601 · immutable	Creation timestamp. Immutable after creation.

Product Type → Valid Pricing Models and Entitlement Structures

Product type	Valid pricing models	Required entitlement type	Meter type(s)	Rev rec treatment
token_bundle	per_token · subscription · hybrid	Token Budget entitlement	token (input/output/cache)	Ratable over service period
agent_workflow	per_task · subscription · hybrid	Task Limit entitlement	task (with completion signal)	On task completion or ratable
outcome_service	per_outcome · gain_share · hybrid	Outcome SLA entitlement	outcome (verification-triggered)	Variable consideration — ASC 606
model_access	per_token · per_api_call	Rate Limit entitlement	token or api_call	Ratable as consumed
compute_reservation	reservation · spot · on_demand	Capacity Reservation entitlement	time (per minute or hour)	Ratable over reservation period
platform_subscription	subscription · freemium	Seat entitlement	user or flat	Ratable monthly/annually

composite	hybrid (components array)	Multiple — one per component	Multiple — one per component	Allocation across obligations

FOR THE PRODUCT MANAGER

Version policy must be decided before launch, not after

The most common versioning mistake: treating every model improvement as a version update that creates a new Product object. This rapidly produces an unmanageable catalog with dozens of nearly-identical products and complex entitlement migration logic. Establish a versioning policy at launch: a new Product is created when the commercial terms change significantly (new pricing model, new SLA commitment, fundamentally different capability). A version increment on the existing Product is used for improvements that do not change the commercial relationship. Document the policy and enforce it through catalog governance.

Object 2 — Price

The Price object defines the commercial terms under which a Product is available for purchase. The relationship between Product and Price is deliberately one-to-many: a single Product can have multiple active Price objects simultaneously, representing different tiers, different markets, different customer segments, and different contract vintages.

This one-to-many relationship is one of the most commonly misunderstood aspects of the monetization data model. Many practitioners instinctively model price as a field on the Product object — the product has a price. This works for simple, single-tier products with no pricing variation. It fails entirely for AI products, which routinely have:

Multiple pricing tiers based on consumption volume. A per-token price that is lower for customers consuming more than 10 billion tokens per month than for customers consuming less than 1 billion.

Multiple customer segments with different commercial terms. An enterprise price that includes a subscription floor, and a self-service price that is pure consumption-based with no minimum.

Historical prices that must be preserved for billing in-flight contracts. When a price changes, the new price applies to new contracts. Contracts signed under the old price continue to be billed at the old price until renewal. Both prices must be active simultaneously for billing to work correctly.

The Price object's type field is the most consequential field in the object, because it determines which downstream systems the price can integrate with. A per_token price requires token-level metering data from the Meter object. A per_task price requires task completion events from the Event taxonomy. A per_outcome price requires verified outcomes from the Outcome object. Each type creates specific data dependencies that must be satisfied for billing to work.

The hybrid price type — a price that combines multiple pricing dimensions simultaneously — is the most architecturally complex and the most commercially important for mature AI deployments. A hybrid price that combines a monthly subscription fee with per-token overage pricing and per-outcome achievement bonuses requires a price object that contains three components, each with its own rules, and an aggregation methodology that determines how the components combine into a single invoice line item.

Price Object — Core Fields and Type-Specific Extensions

Field	Type	Description / Values
`id`	UUID v4 · required	Globally unique. Multiple Price objects can reference the same Product.
`product_id`	UUID · required	Reference to the Product this price governs. Validated — must be an existing Product.
`type`	enum · required	per_token | per_task | per_outcome | subscription | reservation | revenue_share | hybrid
`currency`	ISO 4217 · required	e.g. USD, EUR, GBP. All amounts in this price object use this currency.
`effective_from`	ISO 8601 date · required	Date from which this price applies. Historical prices never deleted — deprecated instead.
`status`	enum · required	draft | active | deprecated — active prices cannot be deleted, only deprecated

— per_token extensions —		
input_token_price	decimal · conditional	Price per million input tokens. Required if type = per_token.
output_token_price	decimal · conditional	Price per million output tokens. Required if type = per_token. Typically 2–5× input price.
cache_multiplier	decimal · optional	Multiplier for cached input tokens. Typically 0.1 (10% of standard input price).
— per_task extensions —		
task_price	decimal · conditional	Price per completed task. Required if type = per_task.
task_definition	string · conditional	Precise definition of a billable task. Required. Must be specific enough to adjudicate disputes.
— per_outcome extensions —		
outcome_price	decimal · conditional	Price per verified outcome. Required if type = per_outcome.
outcome_definition	string · conditional	Precise outcome definition. Required. Must specify metric, threshold, measurement source.
verification_method	string · conditional	How outcome verification occurs. Required. e.g. 'customer CRM webhook', 'third-party audit'
— subscription extensions —		
period	enum · conditional	monthly | annual | multi_year. Required if type = subscription.
seats_included	integer · conditional	Included seats/users. 0 = unlimited. Required if type = subscription.
overage_price	decimal · optional	Per-unit price for consumption above included allocation. Enables hybrid billing.
— hybrid extension —		
components	Price[] · conditional	Array of complete Price specifications, one per pricing dimension. Required if type = hybrid.
volume_tiers	tier[] · optional	Array of {min_qty, max_qty, price}. Applied when quantity exceeds tier threshold.

⚠ The Price Snapshot Problem

When a Price object is updated (e.g., a per-token rate decreases), the new price must not be applied to contracts that were signed under the old price. The correct architecture: the Entitlement object carries a price_id reference that locks the price at contract signing. The billing engine always uses the price referenced in the Entitlement, never the current active price for the product. Failing to implement this correctly causes one of the most damaging billing errors possible: systematically over- or under-billing every customer on the affected product until the bug is caught.

Object 3 — Entitlement

The Entitlement object is the operational heart of AI monetization. It governs whether a customer's request for AI service should be honored at any given moment. It tracks how much of a customer's committed allocation has been consumed. And it is the object through which financial controls — token budgets, task limits, spending caps — are enforced in real time.

Most practitioners underestimate the Entitlement object's importance because they conflate it with the Contract. The Contract says what the customer has bought. The Entitlement says whether they can use it right now. These are different questions with different operational implications.

A customer who has a valid Contract but whose Entitlement has been suspended due to a payment failure should not be able to consume AI services. A customer who has a valid Contract and an active Entitlement but whose token budget has been exhausted at the organizational level should not be able to consume additional tokens without a budget top-up or override. A customer whose Entitlement expires because the contract term ended should not be able to continue consuming services even if the renewal process is in progress.

The Entitlement object's enforcement_policy field — hard_limit, soft_limit_with_alert, or soft_limit_with_approval — is one of the most strategically important configuration decisions in an AI deployment. A hard limit stops consumption when the budget is exhausted. This protects the vendor's economics but risks disrupting the customer's

operations. A soft limit with alert notifies the customer and continues service. This protects the customer's operations but risks margin if consumption runs substantially above budget. The policy choice should be made explicitly and documented in the Entitlement object, not defaulted to whatever the billing platform happens to do.

The Entitlement object also carries the consumption state — how much of the budget has been used — which must be updated atomically with each consumption event. The atomic update requirement is a performance constraint that shapes the technical architecture of the metering system. At high consumption volumes, the contention on Entitlement objects can become a bottleneck. This is a technical constraint with commercial consequences: if the Entitlement check becomes slow, either the AI service latency increases (bad for the product) or the check is deferred (bad for governance). Designing the Entitlement architecture to handle the expected consumption volume is an engineering decision that belongs in the product architecture review.

Entitlement Object — Field Reference

Field	Type	Description / Values
`id`	UUID v4 · required	Globally unique. The primary key for access control checks.
`customer_id`	UUID · required · immutable	The customer this entitlement belongs to. Cannot be changed after creation.
`contract_id`	UUID · required · immutable	The contract that authorizes this entitlement. Source of truth for terms.
`product_id`	UUID · required · immutable	The product the customer is entitled to consume.
`price_id`	UUID · required	The price in force for this entitlement. Locked at contract signing.
`status`	enum · required	pending_activation | active | suspended | expired | cancelled
`activated_at`	ISO 8601 · required when active	Timestamp of activation. Set by provisioning.completed event.
`expires_at`	ISO 8601 · optional	Null for evergreen entitlements. Contract term end date for fixed-term.

enforcement_policy	enum · required	hard_limit | soft_limit_with_alert | soft_limit_with_approval
— token_bundle specific —		
token_budget	integer · conditional	Total tokens allocated for the period. Required for token_bundle products.
tokens_consumed	integer · computed	Running total. Updated atomically with each token.consumed event.
tokens_remaining	integer · computed	token_budget minus tokens_consumed. The real-time governance number.
reset_policy	enum · conditional	none | monthly | annual | rolling_30d — must match the Contract commitment
— agent_workflow specific —		
task_limit	integer · conditional	Maximum tasks per period. Required for agent_workflow products.
tasks_completed	integer · computed	Running count. Updated atomically with each task.completed event.
concurrent_agent_limit	integer · optional	Maximum simultaneously running agent instances. Enforced by agent scheduler.
— outcome_service specific —		
sla_definition	object · conditional	Complete SLA specification: metrics, thresholds, measurement periods, remedies.
outcomes_committed	integer · optional	Outcomes committed per period. Drives SLA breach calculation.
outcomes_delivered	integer · computed	Running count of verified outcomes. Updated by outcome.verified events.

Enforcement Policy Decision Tree

Is this an internal enterprise deployment with a CFO who needs hard spending controls?	Use hard_limit. The risk of service disruption is preferable to the risk of budget overrun. Pair with budget_warning alerts at 80% to give users time to request a top-up.
Is this an external customer deployment where service interruption would damage the relationship?	Use soft_limit_with_approval. Service continues when the budget is hit; an approval workflow is triggered for a budget increase. Requires an SLA commitment on approval turnaround time.

Is this a developer or startup account where you want to prevent accidental runaway costs?	Use hard_limit with a generous initial budget and easy self-serve top-up. The hard limit prevents surprises; the self-serve top-up prevents frustration.
Is this an enterprise account with a dedicated CSM and a mature AI deployment?	Use soft_limit_with_alert. Alert the CSM and the customer's FinOps contact at 80% and 95%. The CSM manages the budget conversation proactively before the limit is reached.

Chapter Two — The Essentials

› The Product type field determines which pricing models are valid — do not mix types that are architecturally incompatible.

› One Product, many Prices: the one-to-many relationship is not optional — AI billing requires it for tier management and price history.

› The Entitlement price_id locks the price at contract signing — billing must use this reference, never the current active price.

› The enforcement_policy on the Entitlement is a strategic decision with customer experience implications — decide it explicitly.

› Entitlement state transitions must be automated — manual state management is the most common source of entitlement errors.

CHAPTER THREE

Object Deep Dives — Meters, Events, and Invoices

The measurement and billing objects. Where consumption becomes revenue.

Object 4 — Meter

The Meter object defines the measurement rules for AI consumption. It is the specification of how raw usage signals — API calls, token counts, task completions, outcome verifications — are translated into billable quantities according to the pricing rules established in the Price object.

The most important design principle for Meter objects is that they must be explicitly designed for each Product-Price combination, not inherited from a template. A per-token price applied to a foundation model API requires a token Meter that counts input tokens, output tokens, and optionally cached input tokens with their respective multipliers. A per-task price applied to an agent workflow requires a task Meter that counts completed workflow executions, with explicit definition of what constitutes a completion and how incomplete executions are handled. These are not the same Meter with different parameters. They are fundamentally different measurement instruments.

The reset_period field on the Meter object governs how the consumption counter resets across billing periods. This field seems simple but has significant operational implications. A monthly reset_period means that a customer who consumed 900,000 tokens in January and has a 1,000,000 token monthly budget starts February with 1,000,000 tokens available again. A contract_period reset means the customer's total budget for the entire contract term is 1,000,000 tokens, and the 900,000 consumed in January reduces what is available for February. These are very different commercial commitments. The Meter's reset_period must match the commitment made in the Contract, or the billing will misrepresent the commercial relationship.

Composite meters — meters that track multiple consumption dimensions simultaneously — are required for any product that bills on more than one axis. An agent workflow product that bills per task plus per token consumed per task (to capture the variable cost of more complex tasks) requires a composite meter with two components: a task component that counts completions and a token component that sums token consumption attributed to each task. The composite meter's combination_rule determines how the two components aggregate into the final billable quantity.

Meter Object — Field Reference

Field	Type	Description / Values
`id`	UUID v4 · required	Globally unique meter identifier.

product_id	UUID · required	The product this meter measures consumption for.
type	enum · required	token | task | outcome | time | api_call | composite
unit	string · required	Human-readable: 'tokens', 'tasks', 'hours', 'API calls'. Appears on invoices.
precision	integer · required	Decimal places to track. 0 for integer counts. Up to 6 for fractional quantities.
aggregation_method	enum · required	sum (default) | max | average | percentile_95
reset_period	enum · required	never | hourly | daily | monthly | annually | contract_period
— token meter extensions —		
input_multiplier	decimal · optional	Applied to input token counts before billing. Default: 1.0
output_multiplier	decimal · optional	Applied to output token counts. Default: 1.0. Vendors often set 2.0–4.0.
cache_multiplier	decimal · optional	Applied to cached input tokens. Default: 0.1 (reflects compute savings).
— task meter extensions —		
task_definition	string · required for task	Precise specification of a metered task. Must exactly match the Price object's task_definition.
completion_signal	string · required for task	The system event name that triggers task completion recording.
partial_task_handling	enum · required for task	ignore | pro_rate | full_charge — policy for incomplete task attempts.
— outcome meter extensions —		
outcome_definition	string · required for outcome	The agreed outcome definition. Must exactly match the Price object's outcome_definition.
verification_source	string · required for outcome	System or process providing outcome verification. Must be agreed by both parties.
verification_latency_hours	integer · required for outcome	Max hours between outcome delivery and verification event. Defines billing event timing.
— composite meter extension —		

components	Meter[] · required for composite	Array of component meter specs. Each is a complete Meter definition.
combination_rule	string · required for composite	How components aggregate to billable quantity. e.g. 'task_fee + (token_count × token_rate)'

> **FOR THE REVOPS ARCHITECT**
>
> **The reset_period must be contractually verified before configuration**
>
> The single most expensive Meter configuration error is a reset_period mismatch with the Contract. A customer who believes their 10M token budget resets monthly, but whose Entitlement has reset_period = contract_period, will consume their entire allocation in two months and be unable to understand why their tokens are exhausted. Before configuring any Meter, verify the reset_period explicitly against the signed contract. Do not infer it from the billing frequency or the subscription period — they are different commercial commitments.

Object 5 — Event

The Event object is the atomic unit of commercial truth in the AI economy. Every token processed, every task completed, every outcome verified, every invoice issued, every payment received — each is recorded as a typed Event with a canonical structure and an immutable identity.

The immutability of the Event object is the foundation of the audit trail. Once an event has been created and written to the event store, it cannot be modified. Corrections are made through Adjustment events that reference the original event's ID, creating a traceable correction history rather than an undetectable overwrite. This immutability is what makes the event store the authoritative record — the source of truth to which all billing calculations and all dispute resolutions are ultimately traceable.

The attribution fields on the Event object — customer_id, product_id, entitlement_id — are what allow events to be connected to their commercial context. These fields are validated at ingestion: an event with a customer_id that does not match a valid customer, or with an entitlement_id that references an entitlement that does not cover the product_id, is flagged as an attribution anomaly. Attribution anomalies are events

that cannot be billed because they lack a valid commercial context. They are the most common root cause of revenue leakage.

The session_id field deserves special attention for agent workflow products. An agent executing a multi-step workflow generates many events — token consumption events for each model call, potentially tool use events for each external API call, a task completion event when the workflow finishes. Grouping these related events with a shared session_id allows the billing system to aggregate them correctly and trace the complete cost of a single workflow execution for P&L-by-customer analysis.

Event Object — Core Field Reference

Field	Type	Description / Values
id	UUID v4 · required · immutable	Globally unique. Used as idempotency key. Never reused. Never modified.
type	string · required	Event type from canonical taxonomy. e.g. 'token.consumed', 'task.completed'
timestamp	ISO 8601 ms · required	Millisecond precision. Records when the event OCCURRED, not when it was processed.
source_system	string · required	Identifier of the system generating this event. Required for attribution audits.
customer_id	UUID · required	The customer whose consumption this records. Validated at ingestion.
product_id	UUID · required	The product being consumed. Validated against customer's active entitlements.
entitlement_id	UUID · required	The entitlement governing this consumption. Validated — must be active and cover the product.
session_id	UUID · recommended	Groups related events from one interaction or agent workflow. Essential for task-level cost attribution.
agent_id	UUID · recommended	The AI agent that generated this event, if applicable. Required for agent commerce audit trail.
model_id	UUID · recommended	The foundation model used. Required for model-level cost attribution and marketplace royalty calculation.
workflow_id	UUID · recommended	For agent workflow events — links to the Agent Task parent.

payload	JSON · required	Type-specific fields. Schema defined by event type. See event taxonomy in Book 4 Appendix B.

Event Attribution Validation — What Is Checked at Ingestion

Field	Validation check	Failure handling	Leakage risk if skipped
customer_id	Must match an existing Customer record	Reject with error: 'Unknown customer'	Event unbillable — revenue lost
product_id	Must match an existing active Product	Reject with error: 'Unknown product'	Event unbillable — revenue lost
entitlement_id	Must be active and cover the customer + product combination	Flag as attribution anomaly; route to resolution queue	Event unattributable — potential revenue loss
timestamp	Must be within the valid submission window (typically ±24h of now)	Flag as late or future event; apply late submission policy	Billing period boundary errors
payload schema	Must validate against the schema for this event type	Reject with schema validation error and field details	Incomplete billing data — aggregation errors possible

Object 6 — Invoice

The Invoice object is the formal commercial document that requests payment for AI services consumed. Its core design requirement is complete traceability: every line item on every invoice must be traceable, through the golden thread, to the specific events that generated it.

The event_summary_uri field on each invoice line item is the traceability mechanism. It provides a URI to a reconciliation report — the detailed breakdown of events that were aggregated to produce that line's charge. This report is the evidence that resolves billing disputes: the vendor can show the customer exactly which events they are being billed for, and the customer can verify those events against their own system logs.

Invoices are immutable after issuance in a well-governed billing system. Corrections are made through InvoiceAdjustment objects that reference the original invoice and create a traceable audit trail of the correction. Never modify an issued invoice directly — the audit trail is broken and the revenue recognition implications become unmanageable.

Invoice Line Item — Required Fields

Field	Type	Description / Values
line_id	UUID · required	Unique identifier for this line within the invoice.
product_id	UUID · required	Product being billed on this line.
price_id	UUID · required	The Price object used to calculate this line. Locked to the price_id in the Entitlement.
entitlement_id	UUID · required	The Entitlement whose consumption is being billed.
description	string · required	Human-readable description of what is being charged. Appears on the customer-facing invoice.
quantity	decimal · required	Billable quantity. Unit matches the Meter's unit field.
unit	string · required	Unit of measurement. e.g. 'tokens (millions)', 'tasks', 'outcomes', 'GPU-hours'
unit_price	decimal · required	Price per unit from the Price object. Locked at billing time.
line_total	decimal · required	quantity × unit_price. Pre-tax. Verified sum.
event_count	integer · required	Number of raw events aggregated into this line. Must be ≥ 1. Auditors check this.
event_summary_uri	string · required	URI to the event-level reconciliation report for this line. The golden thread anchor.
period_start	ISO 8601 · required	Start of the consumption period this line covers.
period_end	ISO 8601 · required	End of the consumption period this line covers.

FOR THE BILLING OPS LEAD

The event_summary_uri must be accessible to the customer within 48 hours of a dispute

The event_summary_uri is only valuable if the customer can access the report it points to. A URI that returns a 403 when the customer tries to access it, or that points to data that has not been indexed yet, is worse than no URI at all — it creates the impression of transparency without

> the substance. Establish and test the event summary generation and hosting pipeline before you issue a single consumption-based invoice. Test the customer-facing access specifically. The dispute you cannot resolve because the event data is inaccessible will damage the relationship more than the charge itself.

Chapter Three — The Essentials
› Meter reset_period must be explicitly verified against the Contract before configuration — mismatches are among the most expensive billing errors.
› Events are immutable after creation. All corrections use Adjustment events with full attribution trail.
› The entitlement_id validation at event ingestion is the primary defense against revenue leakage from unattributed consumption.
› The event_summary_uri on every invoice line is not optional — it is the mechanism that makes billing defensible.
› Invoices are immutable after issuance. Corrections through InvoiceAdjustment objects only.

CHAPTER FOUR

The Remaining Objects and the Relationship Architecture

Credits, allocations, token budgets, agent tasks, outcomes, assets — and how all thirteen connect.

Objects 7–13: Reference Definitions

The remaining eight canonical objects complete the monetization data model.

The Invoice object is the formal commercial document that requests payment. Its defining characteristic for AI billing is the traceability requirement: every line item must reference the events that generated it through the event_summary_uri field. An Invoice that cannot be traced to its source events is a trust liability and an audit risk.

The Contract object is the binding commercial agreement. Its most important function in the data model is as the authoritative source for Entitlement terms and pricing rules. When a contract is amended, the change must flow downstream to update the relevant Entitlements and price references — a data flow that must be explicitly designed and tested, not assumed.

The Credit object records reductions in amount owed. Its authorization_required field — always set to true in a well-governed system — ensures that no credit is applied without documented approval. Credits without authorization are the most common manual control failure in billing operations.

The Allocation object governs revenue recognition for multi-element arrangements. For AI products that bundle multiple performance obligations — a subscription component, a consumption component, and an outcome component — the Allocation object determines how the transaction price is distributed across those obligations for accounting purposes. This object directly affects revenue recognition timing and must be reviewed by the finance team, not just the commercial team.

The Token Budget object is the enterprise financial control instrument for AI consumption. It sits above individual Entitlements and governs aggregate spending across a team, department, or organization. The relationship between Token Budgets and Entitlements is hierarchical: a Token Budget exhaustion can suspend all Entitlements within its scope regardless of individual Entitlement status.

The Agent Task object records a single execution of an autonomous AI workflow. Its relationship to the Event object is parent-to-children: an Agent Task typically generates multiple Events (the token consumption events for each model call within the workflow). The Agent Task is the billing unit for per-task pricing; the Events it contains are the audit trail for that charge.

The Outcome object records a verified business result. Its verified_by field — the system or person that confirmed the outcome occurred — is the attribution anchor for outcome-

based billing. Without a reliable verification source, outcome billing becomes a claims-processing exercise prone to dispute.

The Asset object is the per-customer inventory of AI capabilities currently active. Its primary operational purpose is to support the renewal and expansion motions: the Asset register tells the CSM what the customer has deployed, what is approaching expiry, and where there are natural expansion opportunities.

Remaining Objects — Quick Reference

Object	Primary purpose	Key field	Downstream effect
Contract	Binding commercial agreement	performance_obligations (array)	Authorizes Entitlements · Defines Allocation rules
Credit	Reduction in amount owed	authorized_by (UUID — required)	Reduces Invoice balance · Triggers rev rec adjustment
Allocation	Rev rec distribution across obligations	allocation_method (enum)	Drives revenue recognition timing and amount
Token Budget	Hierarchical AI spend governance	enforcement_policy (enum)	Governs all Entitlements in scope · Triggers approval workflows
Agent Task	Single autonomous workflow execution	completion_signal (string)	Parent of Event children · Billing unit for per_task pricing
Outcome	Verified business result	verified_by (string — attribution anchor)	Triggers billing event · Updates SLA delivery counter
Asset	Per-customer capability inventory	renewal_date (ISO 8601)	Drives renewal workflows · Identifies expansion opportunities

The Relationship Architecture

The relationships between the thirteen objects are as important as the objects themselves. An incorrect relationship — a wrong cardinality, a missing foreign key, an undefined cascade behavior — is as damaging to the data model as an incorrectly defined object.

The primary relationship chain in the monetization data model runs: Contract → Entitlement → Meter → Event → Invoice. A Contract authorizes one or more Entitlements. Each Entitlement is governed by a Meter that defines how consumption is measured. The Meter generates Events that record consumption. Events are aggregated according to billing rules and presented on an Invoice. This chain is the golden thread: every invoice line item should be traceable, through this chain, to the contract that authorized it.

Secondary relationships branch off this primary chain. The Product object is referenced by both the Price object (which defines how the product is charged) and the Entitlement object (which defines what the customer has the right to consume). The Credit object references the Invoice object it reduces. The Allocation object references the Contract whose revenue it distributes across performance obligations. The Token Budget object has a scope relationship to the Entitlements it governs.

The cascade behaviors at relationship boundaries require explicit design. When a Contract is terminated, what happens to its Entitlements? (They should be cancelled, not left active.) When an Entitlement expires, what happens to Meters that reference it? (They should stop accepting events for that entitlement.) When an Invoice is disputed, what happens to the Revenue Recognition entries derived from it? (They should be placed in suspense until the dispute is resolved.) These cascades are not defaults — they are business rules that must be designed and implemented.

Key Object Relationships — Cardinality and Cascade Behavior

Relationship	Cardinality	Cascade on source termination	Data integrity constraint
Product → Price	1 : many	Prices deprecated, not deleted	price.product_id must reference active Product
Product → Entitlement	1 : many	Entitlements cancelled if Product archived	entitlement.product_id must reference active Product
Contract → Entitlement	1 : many	Entitlements cancelled when Contract terminates	entitlement.contract_id must reference active Contract

Entitlement → Meter	many : 1	Meter stops accepting events for this entitlement on expiry	meter.product_id must match entitlement.product_id
Meter → Event	1 : many	Events archived, not deleted	event.entitlement_id must match active Entitlement at ingestion time
Event → Invoice Line	many : 1	Invoice lines preserved as immutable records	line.event_count must equal events attributed to this line
Token Budget → Entitlement	1 : many	Budget exhaustion can suspend all governed Entitlements	entitlement scope must be within budget scope
Agent Task → Event	1 : many	Task cancellation creates task.cancelled event; child events preserved	event.workflow_id must reference valid Agent Task

⚠ The Cascade Design is a Business Decision, Not an Engineering Default

Every relationship in the data model has a cascade behavior when the source object changes state. What happens to Entitlements when a Contract is terminated? What happens to open Invoice Lines when an Entitlement expires mid-period? What happens to in-flight Agent Tasks when a Token Budget is exhausted? These are not technical defaults — they are business rules that must be explicitly decided, documented in the data model specification, and implemented consistently across all systems. Leaving cascade behavior to engineering defaults produces inconsistent behavior that surfaces as billing errors, customer disputes, and audit findings.

The Golden Thread — Tracing Value from Concept to Cash

The golden thread is the unbroken data lineage that connects every commercially significant decision and event from the product's initial definition to the revenue it eventually generates. Tracing it requires that each object in the chain carry a reference to the object it derived from — and that the chain be queryable end-to-end.

The Golden Thread — Complete Object Chain			
Stage	**Object**	**Key outbound reference**	**Traceability question**

1. Product	Product	→ Price (pricing_reference)	What commercial terms govern this capability?
2. Offer	Price	→ Entitlement (via price_id lock)	What are the specific terms for this customer?
3. Contract	Contract	→ Entitlement (contract_id)	What rights does this customer have?
4. Entitlement	Entitlement	→ Meter (product_id match) · → Token Budget (scope)	Can this customer consume right now?
5. Metering	Meter	→ Event (governs structure)	How is this consumption measured?
6. Event	Event	→ Invoice Line (via aggregation)	What billable activity occurred?
7. Invoice	Invoice	→ Event (via event_summary_uri)	What are we charging for and why?
8. Payment	Payment	→ Invoice (invoice_id)	Has the invoice been settled?
9. Recognition	Allocation	→ Contract (contract_id)	How is this revenue recognized under ASC 606?
10. Renewal	Asset	→ Entitlement (entitlement_id)	What is approaching expiry and needs renewal?

> ***"A break in the golden thread is not an inconvenience. It is revenue that was created but not captured — and a liability that will surface as a billing dispute, an audit finding, or a customer trust event."***

Chapter Four — The Essentials

› All seven remaining objects have specific commercial purposes — none is optional for a complete AI billing architecture.

› Cascade behaviors at relationship boundaries are business rules, not engineering defaults — decide them explicitly.

› The golden thread must be queryable end-to-end: from a recognized revenue entry back to the events that generated it.

› The Asset object is the renewal and expansion engine — without it, the post-sale revenue motion is reactive, not proactive.

› The Token Budget's hierarchical scope is the CFO's primary financial control over AI spending — implement it before deployment at scale.

CHAPTER FIVE

The 3C Framework: Compose, Configure, Converse

Three interaction modes for AI-native monetization. How offers are built, governed, and discussed.

The 3C Framework describes the three modes through which commercial intent is translated into commercial reality in an AI-native monetization system. It is a design pattern, not a technology requirement — the three modes can be implemented in many ways, but all three must exist for a complete commercial architecture.

Compose: Building Offers from Objects

The Compose mode is the structured assembly of commercial offers from the canonical monetization objects. A pricing architect in Compose mode is working with Product objects, Price objects, Entitlement structures, and Meter configurations — combining them into an Offer that can be presented to a customer segment.

Composition is where the commercial intent defined by the strategy team is translated into the data structures that the billing system will enforce. A well-composed offer is internally consistent: the Product type, the Price type, the Entitlement structure, and the Meter configuration all refer to compatible objects with no type mismatches. An

inconsistent offer — a composite Product priced as a simple per-token Price with no component specifications — will fail at billing time in ways that may not be caught until the first invoice is generated.

The Compose mode is the domain of product managers and pricing architects. The primary tool is the product catalog configuration interface — the system where new products, prices, and offer bundles are created. The primary governance mechanism is the catalog validation: automated checks that flag type mismatches, missing required fields, and relationship violations before an offer is published.

Compose Mode — Offer Consistency Checklist

Check	Passes when	Fails when	Fix
Product-Price type match	Price type is valid for Product type (e.g., per_task Price for agent_workflow Product)	Price type is invalid for Product type (e.g., reservation Price for outcome_service Product)	Change Price type to match Product type, or create a different Product
Entitlement-Meter alignment	Entitlement structure matches Meter type (e.g., token_budget Entitlement with token Meter)	Entitlement type and Meter type are incompatible	Configure a compatible Meter for this Entitlement type
Composite completeness	Hybrid Price has a component spec for every pricing dimension the Product requires	Hybrid Price is missing a component (e.g., no consumption component for a token_bundle)	Add the missing Price component to the hybrid
SLA-Outcome alignment	Outcome-service Products have an SLA definition in the Entitlement	Outcome-service Products have no SLA — no basis for breach calculation	Add SLA definition before activating the Entitlement
Revenue recognition	Allocation object exists for multi-element arrangements	Multi-element offer has no Allocation — rev rec undefined	Create and validate the Allocation object with finance sign-off

Configure: Setting Rules Via Data

The Configure mode is the real-time governance layer — the policies, rules, and controls that determine what is permitted in the commercial system without requiring human intervention for each decision. Configuration is how the business rules defined in strategy meetings become the automated enforcement in billing systems.

Configuration decisions in an AI monetization system fall into three categories. Pricing configuration sets the rates, tiers, and discounts that determine what each customer pays for each unit of consumption. Entitlement configuration sets the budgets, limits, and enforcement policies that govern what each customer can consume. Governance configuration sets the approval workflows, escalation rules, and alert thresholds that determine what requires human review.

The most important configuration decision in AI monetization is the enforcement policy on Token Budget and Entitlement objects — hard limit versus soft limit. This decision determines what happens at the moment a customer's budget is exhausted, and it has significant implications for both customer experience (hard limit can disrupt workflows) and vendor economics (soft limit can allow significant overconsumption). The configuration should be documented as a conscious policy decision, not left to the default of whatever the billing platform provides.

Converse: Natural Language Pricing

The Converse mode is the interface between the monetization system and natural language — the ability for AI agents, customers, and operators to interact with pricing and billing data through conversation rather than through structured forms.

Three Converse applications are commercially significant. The first is agent-initiated price discovery: an AI agent that is about to purchase a service can query the catalog in natural language — 'What does it cost to review 500 contracts using the legal AI?' — and

receive a structured price estimate without requiring the agent to parse a pricing table. The Catalog API's search endpoint implements this. The second is customer-facing billing explanation: a customer who does not understand a charge on their invoice can ask the billing system in natural language — 'Why was I charged $4,200 on line 3 of my October invoice?' — and receive a plain-language explanation with supporting evidence. The MCP server's explain_invoice tool implements this. The third is operator-facing configuration: an operations administrator can configure pricing rules through natural language instructions — 'Add a 15% volume discount for customers consuming more than 100 million tokens per month' — with the system translating the instruction into the appropriate Price object configuration. This is the most nascent of the three applications but represents the direction of AI-native commercial operations.

The 3C Framework — Summary	
Compose	Structured assembly of offers from monetization objects. Ensures internal consistency of commercial products before they reach customers. Domain of product managers and pricing architects.
Configure	Real-time governance rules that automate commercial policy enforcement. Pricing rates, budget limits, approval thresholds, alert triggers. Domain of RevOps and billing operations.
Converse	Natural language interface to pricing and billing data. Agent-initiated service discovery, customer billing explanation, operator rule configuration. Domain of the MCP server and AI-native tooling.

Chapter Five — The Essentials

› The 3C Framework describes three interaction modes: Compose (building offers), Configure (setting rules), Converse (natural language interfaces).

› Compose mode produces consistent offers by enforcing compatibility between Product type, Price type, Entitlement structure, and Meter configuration.

› Configure mode automates business rules that would otherwise require human intervention — enforcement policies, approval thresholds, alert triggers.

› Converse mode makes the commercial system accessible to AI agents and natural language queries — the foundation of agent commerce.

› All three modes must exist in a complete AI monetization architecture. Missing any one creates gaps that surface as manual workarounds.

PART TWO

Pricing the Five Layers

Implementation-level guidance for each layer of the AI economy stack.

CHAPTER SIX

Compute Economy: GPU-as-a-Service Billing

Per-GPU per-minute, reservation economics, cost floors, and AI factory pricing.

GPU-as-a-Service billing is the simplest layer of the AI economy to understand and among the most complex to implement correctly at scale. Simple, because the fundamental pricing logic is straightforward: you reserve capacity, you use capacity, you pay for capacity. Complex, because the volume of billing events, the variety of pricing structures, and the financial significance of utilization rates create operational challenges that are easy to underestimate.

This chapter covers the implementation of compute-layer pricing for practitioners who are building the billing architecture for AI infrastructure products, or who are responsible for the FinOps governance of compute spending in an enterprise AI deployment.

The Three Compute Pricing Models

The three primary pricing models at the compute layer are on-demand, reserved capacity, and spot.

On-demand pricing charges for actual compute consumption with no upfront commitment. The unit is typically the GPU-hour or the instance-hour: you pay for each hour that a GPU or compute instance is active, prorated to the minute or second for partial hours. On-demand pricing is the most flexible — no commitment, pay only for what you use — but also the most expensive per unit. It is appropriate for variable, unpredictable workloads where utilization is difficult to forecast.

Reserved capacity pricing offers a significant discount — typically 30–60% versus on-demand — in exchange for a commitment to a defined capacity for a defined period, usually one year or three years. The customer pays for the reserved capacity whether they use it or not. The vendor benefits from the revenue predictability and can plan infrastructure investment accordingly. Reserved pricing is appropriate for stable, predictable workloads where the capacity requirement is known in advance.

Spot pricing sells unused capacity at a steep discount — often 70–90% below on-demand — with the caveat that the capacity can be reclaimed by the vendor with short notice when demand from higher-priority workloads requires it. Spot pricing is appropriate for batch workloads that are tolerant of interruption: training runs, offline inference, data processing jobs. It is not appropriate for interactive AI services where latency and availability are commitments.

The billing implementation for each model requires different event granularity. On-demand billing requires per-instance start and stop events, from which billing duration is calculated. Reserved capacity billing requires a reservation record (the committed capacity) and a utilization record (the actual usage), from which both the base charge (the reserved amount) and the overage charge (usage above the reservation) are calculated. Spot billing requires the same events as on-demand, plus reclamation events that interrupt billing when capacity is reclaimed.

A critical implementation consideration for compute billing is the relationship between billing granularity and billing accuracy. If compute instances are billed by the hour, a customer who runs a GPU for 61 minutes pays for two hours — a 49% overcharge relative to actual usage. Most enterprise AI customers will eventually notice and dispute this. Billing to the minute or second requires more event granularity but is significantly more accurate and significantly less likely to generate disputes.

Compute Pricing Models — Implementation Reference

Model	Pricing unit	Billing event granularity	Use case	Margin profile
On-demand	Per GPU-hour or instance-hour	Start event + Stop event per instance (prorate to minute)	Variable, unpredictable workloads — experiments, burst capacity	Highest per-unit revenue, highest per-unit cost — thin margin
Reserved	Committed capacity × period price	Monthly invoice for reserved amount + on-demand overage meter	Stable production workloads with known baseline requirement	Predictable revenue, amortized fixed cost — better margin at high utilization
Spot	Market-rate per GPU-hour	Start event + Stop/Preemption event (prorate to second)	Batch jobs, training runs, offline inference — interruption tolerant	Variable — margin depends on utilization of otherwise idle capacity
Hybrid reservation+spot	Reserved floor + spot for peak	Reserved invoice + spot meter for excess	Workloads with predictable base + variable peak demand	Best overall: stable floor margin + opportunistic spot revenue

Three pricing structures illustrate the range of compute billing implementations in current practice.

A reserved GPU cluster for enterprise AI inference: the customer reserves 8 H100 GPUs for 12 months at a committed price per GPU per hour. The reservation creates an

Entitlement object with the reserved capacity specification. Monthly billing generates an Invoice line for the reserved capacity charge (8 GPUs × price × hours in the month) plus a separate line for any on-demand GPU usage above the reservation. The Meter tracks both reserved utilization and on-demand usage separately to support the two-part billing.

AI factory pricing for a large-scale training run: the customer books compute capacity for a 30-day training run. The billing is based on actual GPU-hours consumed, measured at 1-minute granularity, with a minimum daily charge to cover infrastructure fixed costs. The Meter captures start and stop events for each GPU in the training cluster, aggregates at the day level for daily billing visibility, and aggregates at the period level for the monthly Invoice.

Shared inference infrastructure with chargeback: an enterprise deploying AI across multiple business units wants to charge each unit for its share of shared GPU infrastructure. The billing system attributes each inference request to the requesting business unit through the agent_id or team_id field on the Event object. A monthly allocation report shows each business unit's share of total GPU consumption, which is used for internal chargeback. The external Invoice is for the total infrastructure cost; the chargeback allocation is an internal reporting function, not a separate external billing event.

> **FOR THE REVOPS ARCHITECT**
>
> **Billing granularity is a trust decision, not just a precision decision**
>
> Hourly billing for GPU compute means a customer who runs a GPU for 61 minutes is charged for 2 hours — a 97% overcharge on that marginal minute. At small scale this is tolerable. At enterprise scale (thousands of GPU-hours per month), the cumulative overcharge on fractional hours becomes significant enough to generate disputes and audit attention. Invest in per-minute or per-second billing granularity before you reach enterprise scale. The engineering cost is modest; the trust benefit is substantial.

Reservation Economics — The Pricing Architect's Model

Reserved capacity pricing requires the pricing architect to model three economic relationships simultaneously: the vendor's break-even utilization (the utilization rate at which reserved pricing covers its costs), the customer's optimal commitment level (the quantity that minimizes their total cost of ownership), and the market-clearing discount rate (the discount that makes reservation attractive relative to on-demand without pricing below the vendor's break-even).

The vendor's break-even utilization for a reserved GPU is approximately: (reserved price per hour) / (on-demand price per hour). If the vendor charges $1.50/hour for reserved and $2.50/hour for on-demand, break-even utilization is 60%. Below 60% utilization, the reserved pricing is not covering costs. Above 60%, the vendor is profitable on the reservation. The reservation discount (40% in this example) was calibrated to achieve break-even at a utilization level the vendor expects to achieve.

The customer's optimal commitment level is the point at which the cost of reserving one additional unit equals the expected cost of purchasing that unit on-demand. Customers who can forecast their utilization accurately and commit at the right level capture the maximum discount. Customers who over-commit pay for capacity they cannot use; customers who under-commit pay on-demand rates for workloads that could have been reserved.

Reservation Discount Calibration Framework	
What is our target break-even utilization rate?	Set this based on the infrastructure economics. If reserved capacity costs 60% of on-demand to run at any utilization, break-even utilization is 60%. Add a margin buffer of 10–15% to set the pricing threshold.
What discount makes reservation compelling for customers?	Model the customer's cost at different utilization rates. The discount should make reservation cheaper than on-demand at the utilization rate the average customer achieves.
What term length optimizes the tradeoff between commitment and discount?	Longer terms (3-year vs 1-year) justify deeper discounts because they provide more revenue certainty. But longer terms require more customer confidence in their capacity needs.

How do we handle mid-term capacity needs above the reservation?	On-demand overage is the standard. Design the overage pricing to be higher than the reserved rate to incentivize customers to right-size their reservations upfront.

Chapter Six — The Essentials

› Three compute pricing models: on-demand (flexibility, thin margin), reserved (predictability, better margin at utilization), spot (idle capacity monetization).

› Per-minute billing granularity is a trust investment — hourly billing creates systematic overcharges on fractional hours.

› Reservation discount calibration requires explicit break-even modeling — discount must cover costs at expected utilization, not hoped-for utilization.

› Hybrid reservation-plus-spot is often the optimal structure for mature enterprise deployments with predictable baseline plus variable peaks.

› Chargeback for shared AI factory infrastructure requires event-level team/unit attribution — the team_id or agent_id fields on events.

CHAPTER SEVEN

Model Economy: API Metering and Model-as-a-Service

LLM API pricing, fine-tune royalties, model marketplace economics, and the price update problem.

Model-as-a-Service pricing — the API metering and billing that governs access to foundation models — is the most competitive and fastest-evolving pricing environment in the AI economy. Prices have fallen dramatically since 2023 and will continue to fall as model efficiency improves and competition intensifies. The pricing architecture built today must be designed to accommodate significant price changes without requiring a wholesale re-implementation.

This chapter covers the implementation of model-layer pricing from two perspectives: the model vendor building the billing system, and the enterprise buyer building the FinOps governance for API consumption.

Per-Token Pricing: The Universal Standard

Per-token pricing is the universal standard at the model layer. The universal structure: separate prices for input tokens and output tokens, with output tokens typically priced 2–5 times higher than input tokens because generation is computationally more expensive than context processing. Optionally: a lower price for cached input tokens (tokens that were already in the model's context cache and do not require re-processing), a higher price for extended context (prompts that exceed the standard context window length), and premium pricing for specific capabilities (function calling, tool use, structured output).

The implementation complexity at the model layer is not in the pricing logic — it is in the metering infrastructure. A high-volume API service may process millions of requests per minute, each generating a token consumption event. The metering system must:

Handle the volume without becoming a latency bottleneck for the API response. The token count for an API call can be calculated synchronously (the model knows how many tokens were processed) and written to the event queue asynchronously after the response is returned. The billing calculation does not need to be complete before the response is delivered.

Deduplicate events for retried requests. When a client retries an API call because it did not receive a response (perhaps due to a network failure), the model may have actually processed the first request and the response was lost. Without deduplication, the customer is billed twice for a request that was only useful once. The event_id field serves as the idempotency key: the same request_id generates the same event_id, and a duplicate submission is discarded rather than billed twice.

Attribute events to the correct commercial context. A single organization may have multiple API keys, each associated with a different product tier, a different billing account, or a different cost center. The event attribution must correctly identify the organizational account, the specific API key's associated price and entitlement, and the relevant cost center for chargeback purposes.

The price update problem is particularly acute at the model layer, where prices change frequently as efficiency improves and competition intensifies. The data model must support the simultaneous existence of multiple Price objects for the same Product — the old price for in-flight contracts, the new price for new contracts — without any system applying the wrong price to the wrong contract. The effective_from field on the Price object, combined with the price_id reference on each Entitlement, ensures that billing always applies the correct price for each specific contract regardless of when the price was updated.

Token Pricing Dimensions — Implementation Reference

Dimension	Description	Billing event field	Typical ratio vs input	Implementation note
Input tokens	Tokens in the prompt: system instructions + context + user message	payload.input_tokens	1× (baseline)	Count before model processes — known when request is submitted
Output tokens	Tokens generated by the model in its response	payload.output_tokens	2–5× input price	Count after generation completes — billing event sent post-response
Cached input tokens	Input tokens served from context cache — previously processed	payload.cached_input_tokens	0.1–0.2× input price	Requires cache hit detection in inference infrastructure

Extended context premium	Surcharge for prompts exceeding the standard context window	payload.input_tokens flagged	25–50% premium on excess	Requires context window threshold configuration in Price object
Tool use / function calls	Additional charge for structured output generation	payload.tool_use_count	Variable — often 25% of base input price	Requires tool use detection in the model serving layer

The model marketplace structure — where a platform operator aggregates multiple model providers and takes a revenue share — requires additional data model complexity beyond single-vendor API billing.

The platform operator needs to track the relationship between a customer's request and the specific model provider that fulfilled it, because the revenue share calculation depends on which provider earned the revenue. The Event object's model_id field serves this purpose: each token consumption event records the model that was used, allowing the billing system to aggregate revenue by provider for the royalty calculation.

The royalty split — the percentage of each customer payment that flows to the model provider — is defined in a revenue share agreement that must be represented in the data model. The Allocation object is the appropriate home for this: a marketplace revenue allocation that specifies, for each model provider, the percentage of revenue from requests served by their models that flows to them after the platform fee.

Royalty accounting at the model marketplace level is subject to the same ASC 606 principal-versus-agent analysis that applies to any two-sided marketplace. If the platform is a principal — it controls the service and is responsible for its delivery — then revenue is recognized gross and the model provider payment is a cost of revenue. If the platform is an agent — it merely facilitates the transaction between the customer and the provider — then only the platform fee (net revenue) is recognized. This accounting determination affects the revenue recognition Allocation object's configuration and must be made in consultation with the finance function.

Model Marketplace Revenue Allocation — Required Fields		
Field	**Type**	**Description / Values**
`platform_fee_percentage`	decimal · required	The platform operator's take rate. Applied to gross revenue before provider royalty.
`provider_royalty_percentage`	decimal · required	The model provider's share of net revenue (after platform fee).
`model_id`	UUID · required	The model whose revenue this allocation governs. Must match event.model_id.
`accounting_treatment`	enum · required	gross_principal | net_agent — determines whether revenue is recognized gross or net.
`royalty_calculation_period`	enum · required	monthly | quarterly — frequency of royalty settlement to model providers.

⚠ The Price Update Problem at Scale

At high API volume, the moment a per-token price changes, you have in-flight requests that were initiated under the old price and will complete under the new price. The correct behavior: the price at the time of the request initiation governs the billing, not the price at the time of billing. This requires recording the applicable price_id on each event at the time of ingestion, not at the time of aggregation. Billing systems that look up the 'current' price during aggregation rather than using the price captured on the event will produce incorrect bills during any price transition period.

Chapter Seven — The Essentials

› Per-token pricing has five dimensions: input, output, cached input, extended context, and tool use — each requires a separate billing event field.

› Event-time price locking: record the price_id on the event at ingestion, not at aggregation time, to handle price transitions correctly.

› Deduplication on event_id is mandatory — retried API calls generate duplicate events that must be idempotently discarded.

› Model marketplace royalty accounting requires explicit principal-versus-agent analysis for revenue recognition treatment.

› Fine-tuning royalties require a separate royalty calculation methodology — the base royalty plus a revenue share on fine-tuned model usage.

CHAPTER EIGHT

Token Economy: Pricing the Atom of AI Consumption

Token budget governance, the Token Factory, chargeback models, and consumption optimization.

The token economy is where enterprise AI spending currently concentrates — and where the gap between financial visibility and financial reality is widest. Most enterprises have deployed AI products that consume tokens at varying rates across teams, workflows, and models, without any systematic visibility into what those tokens cost, who is consuming them, or whether the consumption is generating commensurate value.

Building the Token Factory — the organizational capability to govern, optimize, and financially manage token flows — is among the highest-priority FinOps investments for any enterprise with significant AI deployment. This chapter covers the architecture of that capability from the ground up.

Token Pricing Architecture for Application Products

Token pricing for AI products has three standard dimensions that every pricing architect must understand and implement correctly.

Input token pricing covers the tokens in the prompt: the system instructions, the conversation history, the user message, and any context documents included in the request. Input tokens are priced at the base input rate. Most models process input tokens at lower computational cost than generating output, which is why input prices are typically lower.

Output token pricing covers the tokens generated by the model in its response. Output generation is computationally more expensive than input processing because each

output token requires a forward pass through the model. This is why output prices are 2–5 times higher than input prices in most current model APIs.

Cached input token pricing is a pricing dimension that reflects the efficiency gains from caching: when a long context (system prompt, conversation history) has already been processed in a recent request and is available in the model's context cache, re-processing it costs significantly less than the first processing. Vendors that offer caching pass some of this saving to customers through a cached input token price that is typically 10–20% of the standard input token price.

For a pricing architect designing the Token pricing structure for an AI application product, the critical implementation question is whether to expose token pricing directly to customers or to abstract it into higher-level pricing units. Exposing token pricing directly is accurate and transparent, but requires customers to understand token economics — which many enterprise buyers do not. Abstracting token pricing into per-query, per-document, or per-workflow pricing is more intuitive but requires the pricing architect to model the expected token consumption per abstracted unit carefully, building in enough margin to cover token cost variance.

The hybrid approach — exposing token pricing within a consumption tier structure — is increasingly common for mature AI products. The customer purchases a tier (e.g., Standard, Professional, Enterprise) that includes a defined token allocation. Within that allocation, the product handles token consumption internally. Above the allocation, consumption-based overage pricing applies. This structure gives enterprise customers the budget predictability they need while maintaining the consumption alignment that drives fair pricing.

Token Abstraction Strategies — Tradeoffs

Strategy	How it works	Best for	Risk
Direct token pricing	Expose input/output/cache prices directly to customer	Technical buyers, API products, developer tools	Non-technical buyers cannot model their costs; invoice confusion

Per-query abstraction	Define a 'query' as the billing unit; average token cost per query type baked into the price	Products with predictable, uniform interactions	Token cost variance per query creates margin risk if distribution shifts
Tier-based allocation	Monthly token allocation per tier; overage priced per token above allocation	Enterprise buyers needing budget certainty	Allocation sizing requires accurate consumption modeling; wrong size → churn or underpricing
Per-document / per-workflow	Price per document reviewed, per workflow run; token cost baked in	Task-oriented products with predictable document/workflow cost profiles	Task cost variance requires regular repricing as model efficiency changes
Hybrid: tier + task	Subscription tier for base access + per-task charges for agent workflows	Products with both interactive and agentic components	Billing complexity; customer needs two mental models

Building the Token Factory

Token governance is the organizational discipline of managing AI token consumption with the same financial rigour applied to any other significant operational expense. It is the difference between an AI deployment that the CFO can understand, approve, and manage, and one that generates unpleasant surprises on the IT budget line.

The Token Budget object is the data model instrument for governance. Its hierarchical structure — an organizational budget at the top, with team budgets and workflow budgets below — allows governance to be applied at the appropriate level. An engineering organization might have a monthly token budget of 50 million tokens, with sub-budgets of 10 million per team and 1 million per major workflow. The hierarchical enforcement means that a single workflow that runs unexpectedly hot is throttled at the workflow budget before it consumes the team budget, which is throttled at the team level before it consumes the organizational budget.

The Token Factory — the organizational unit responsible for token governance — has four operational functions. Visibility: real-time dashboards showing token consumption by team, workflow, model, and time period, against budget. Allocation: the process for

setting and adjusting token budgets based on business priority and demonstrated consumption patterns. Optimization: the systematic identification and implementation of token efficiency improvements — prompt optimization, model selection, caching strategies — that reduce cost without reducing quality. Governance: the enforcement of token budgets through the Entitlement and Token Budget objects, including the escalation process for budget increase requests.

The chargeback model for token governance determines how token costs are allocated to the business units that generate them. A showback model shows each business unit their token consumption without generating actual financial charges — useful for building awareness and driving behavior change without the friction of internal billing. A chargeback model generates actual internal financial transactions — the business unit's budget is debited for its token consumption. Chargeback creates stronger financial accountability but requires more sophisticated internal billing infrastructure and can create organizational tension if units feel the allocation methodology is unfair.

Token Budget Object — Complete Field Reference

Field	Type	Description / Values
`id`	UUID v4 · required	Globally unique budget identifier.
`customer_id`	UUID · required	The organization this budget governs.
`owner_id`	UUID · required	The team, department, or user the budget belongs to. Supports hierarchical budgets.
`parent_budget_id`	UUID · optional	For hierarchical budgets: the parent budget this one rolls up to.
`total_budget_tokens`	integer · required	Total tokens allocated for this period.
`total_budget_dollars`	decimal · required	Dollar equivalent of the token budget. For financial reporting.
`consumed_tokens`	integer · computed	Running total. Updated atomically with each consumption event within scope.
`consumed_dollars`	decimal · computed	Dollar value of consumed_tokens at applicable token prices.

period_start	ISO 8601 · required	Start of the budget period.
period_end	ISO 8601 · required	End of the budget period.
enforcement_policy	enum · required	hard_limit | soft_limit_with_alert | soft_limit_with_approval
alert_threshold_percentage	integer · required	Percentage of budget consumed that triggers a warning alert. Recommended: 80.
projected_exhaustion_date	ISO 8601 · computed	Based on current burn rate. Key dashboard metric for FinOps.
burn_rate_tokens_per_day	decimal · computed	Rolling 7-day average daily consumption. Used for exhaustion projection.

Token Factory Operating Model — Four Functions

Function	What it does	Data required	Organizational owner
Visibility	Real-time dashboards: consumption by team, workflow, model, time period vs budget	Token Budget consumed values · Event data by team_id / workflow_id / model_id	FinOps team or RevOps
Allocation	Process for setting and adjusting budgets based on business priority and consumption patterns	Historical consumption data · Business unit priorities · Budget cycle calendar	Finance + FinOps
Optimization	Identifying and implementing token efficiency improvements without degrading output quality	Prompt performance data · Model comparison data · Caching hit rates	ML Ops + Product
Governance	Enforcing budgets via Entitlement enforcement_policy · Managing escalations and approvals	Token Budget objects · Entitlement states · Approval workflow outcomes	FinOps + Finance

FOR THE FINOPS LEAD

Chargeback requires event-level team attribution — not estimated allocation

> Organizations that implement token chargeback using estimated allocation (e.g., split the total token bill equally across business units) quickly discover that the units consuming the most tokens object strenuously to paying an equal share. Accurate chargeback requires that every event carry a team_id or cost_center_id that identifies the consuming business unit. This requires a decision about where that attribution is set — at the API key level, at the agent level, or at the request level — and consistent enforcement across all AI deployments. Make this architectural decision before the first chargeback invoice, not after.

Chapter Eight — The Essentials

› Token pricing abstraction strategies range from direct token pricing (maximum accuracy) to per-workflow pricing (maximum simplicity) — choose based on buyer sophistication.

› The Token Budget object is the CFO's financial control instrument — implement it with hierarchical scope before enterprise-scale deployment.

› The Token Factory has four functions: visibility, allocation, optimization, governance — all four must be operational for mature AI FinOps.

› Accurate chargeback requires event-level team attribution, not estimated allocation — make the attribution architecture decision early.

› The burn_rate and projected_exhaustion_date fields on the Token Budget are the proactive governance signals — monitor them daily.

CHAPTER NINE

Agent Economy: Monetizing Autonomous Workflows

Per-task and per-workflow billing, task definition precision, multi-agent cost attribution, and the completion signal.

Agent economy pricing is the most commercially challenging layer to implement correctly, because the billing unit — the task or workflow — is defined by commercial agreement rather than by technical measurement. A token is a token: a precisely defined unit of computation with an objective count. A task is whatever the contract says it is.

This definitional dependency creates the primary commercial risk at the agent layer: task definition disputes. A customer who was billed for 10,000 tasks believes their AI completed 8,000 tasks by their definition of "completed." The vendor believes 10,000 tasks were completed by their definition. The contract's task definition is ambiguous. A billing dispute ensues that is difficult to resolve because the authoritative definition does not exist in a form precise enough to adjudicate the disagreement.

Preventing this dispute starts with the task definition in the Price object, which must be written with the precision of a legal contract clause rather than the generality of a marketing description.

Per-Task Pricing: The Four Required Elements

Per-task pricing requires four elements to be precisely defined in the commercial agreement and in the data model:

Task definition: exactly what constitutes a single billable task. Not "a customer service resolution" but "a customer service ticket closed with no escalation to a human agent within the same session, as recorded in the customer service platform." Not "a contract review" but "a contract reviewed and annotated with issues identified, as indicated by a completed review record in the contract management system."

Completion signal: the specific system event that triggers billing for a completed task. The completion signal must be generated by a system that neither party can unilaterally manipulate — ideally the customer's own system of record (CRM, contract management platform, ticketing system) rather than the vendor's AI system. When the completion signal comes from the vendor's system, the customer has reasonable grounds to dispute it.

Partial task handling: what happens when a task is started but not completed. If an agent begins a contract review but the review is interrupted before a completed record is generated, should the task be billed at full price, at a partial rate, or not at all? The answer depends on whether the interruption was due to a vendor system failure (which

should not result in billing), a customer cancellation (which might be billed at a partial rate), or a timeout (which requires a policy decision).

Multi-agent attribution: for workflows where multiple agents collaborate to complete a task, how is the cost attributed? If a research agent and a writing agent collaborate to produce a report that is billed as one task, the vendor's internal cost attribution must work correctly even though the customer-facing billing is for a single task. This internal attribution matters for P&L-by-customer analysis and for pricing future similar workflows.

Task Definition Quality Checklist

Element	Good example	Bad example	Dispute risk of bad example
Task definition	'Customer service ticket closed by AI with no human escalation within the session, ticket status = resolved in ServiceNow'	'A successfully resolved customer service interaction'	High — 'successfully' and 'interaction' are undefined; every contested resolution becomes a dispute
Completion signal	'ServiceNow webhook event: ticket.status_changed to resolved with human_escalated = false'	'AI system records the ticket as resolved'	High — vendor's own system is not a neutral completion signal; customer can dispute any count
Partial task handling	'No charge for tickets where AI escalates to human within same session; full charge for AI-resolved tickets regardless of CSAT score'	'Partial billing for partially completed tasks'	Very high — 'partial' is undefined; every interrupted interaction is a potential dispute
Multi-agent attribution	'Single task price covers all agent activity within one session_id, regardless of how many sub-agents were invoked'	'Each agent invocation is a billable task'	High — customers often don't know how many sub-agents their workflow uses

Per-workflow pricing — a superset of per-task pricing where the billing unit is a more complex, multi-step business process rather than a single discrete task — introduces additional complexity in the definition and measurement of the billable unit.

A workflow is typically defined in terms of input, process, and output: what the customer provides to initiate the workflow, what the AI does during the workflow, and what the customer receives upon completion. The billing event occurs when the defined output is delivered. Unlike a task, a workflow may involve multiple tools, multiple model calls, and potentially multiple human review steps — all of which must be clearly defined to establish when the billable workflow has been completed.

Workflow pricing must also address the question of retry and remediation: if the AI completes a workflow but the output does not meet the customer's quality standard (as defined in the SLA), and the workflow must be rerun or remediated, does the customer pay for the failed workflow, the remediated workflow, both, or neither? The contract must address this explicitly. A common structure is: no charge for workflows that fail before delivering any output; full charge for completed workflows even if the output requires remediation; credit issued if the output fails to meet the contracted SLA.

Tiered workflow pricing — where the price per workflow decreases as volume increases — is common for high-volume workflow deployments. The implementation requires a Meter that tracks workflow completions within the billing period and applies the correct tier price based on cumulative volume. This is the same tiered pricing logic used at other layers, but the workflow completion counting requires the additional complexity of the partial task handling and completion signal definitions described above.

Agent Task Object — Complete Field Reference		
Field	**Type**	**Description / Values**
`id`	UUID v4 · required	Globally unique. The billing unit identifier for per_task pricing.
`agent_id`	UUID · required	The AI agent that executed this task. Required for agent commerce audit trail.

workflow_id	UUID · required	The workflow definition this task is an instance of. Links to task definition in Price object.
customer_id	UUID · required	The customer whose entitlement this task consumes.
entitlement_id	UUID · required	The entitlement governing this task execution. Validated at task initiation.
status	enum · required	queued | running | completed | failed | cancelled
started_at	ISO 8601 · required	When the task began execution.
completed_at	ISO 8601 · optional	When the task reached a terminal state (completed, failed, or cancelled).
tokens_consumed	integer · computed	Total tokens consumed by all Events with this task's session_id.
subtasks	UUID[] · optional	Child Agent Task IDs for orchestrated multi-agent workflows.
completion_signal_received	boolean · computed	Whether the configured completion signal was received. Determines billability.
billing_eligible	boolean · computed	True when: status = completed AND completion_signal_received = true AND partial_task_handling policy permits billing.

FOR THE PRODUCT MANAGER

The completion signal source determines dispute frequency

The completion signal is the most commercially sensitive element of per-task pricing because it determines when billing is triggered. Completion signals sourced from the vendor's AI system are always challengeable by customers — the vendor controls both the service and the billing trigger, which creates a conflict of interest that sophisticated enterprise customers will notice and object to. Completion signals sourced from the customer's system of record (CRM updated by the AI, ticketing system updated by the AI) are much harder to dispute because both parties can see the same data. Design task billing to use customer-system completion signals wherever possible, even if it requires an additional integration.

Chapter Nine — The Essentials

› Per-task pricing requires four precisely defined elements: task definition, completion signal, partial task handling, and multi-agent attribution.

› The completion signal source is the most commercially sensitive element — customer-system signals are more defensible than vendor-system signals.

› Task definition must be specific enough to adjudicate disputes — marketing-quality descriptions create billing-quality disputes.

› The Agent Task object's billing_eligible computed field is the single source of truth for whether a task charge should be generated.

› Multi-agent workflow attribution must be resolved at the session_id level — customers should not pay per sub-agent invocation.

CHAPTER TEN

Outcome Economy: Engineering Value-Based Billing

Outcome definition, SLA construction, verification architecture, variable consideration, and the attribution methodology.

Outcome economy pricing represents the frontier of AI monetization — the model where the maximum value can be captured and where the implementation complexity is correspondingly highest. This chapter covers the complete implementation of outcome-based pricing: the data model requirements, the measurement infrastructure, the attribution methodology, and the billing event chain from outcome verification to invoice generation.

Outcome pricing is not merely an extension of task pricing. It is a fundamentally different commercial relationship. In task pricing, the vendor charges when the AI completes a defined action. In outcome pricing, the vendor charges when a defined business result is achieved — a result that the AI contributed to but may not have unilaterally caused. This distinction creates the attribution problem that is the central implementation challenge of outcome pricing.

Outcome Definition — The Four-Component Standard

An outcome definition is a precise statement of the business result that triggers billing. The precision requirement is high — imprecise outcome definitions are the single most common cause of outcome billing disputes.

A good outcome definition has four components: the metric being measured (what number or status change defines the outcome), the threshold (what value or state the metric must reach), the measurement source (which system measures the metric and when), and the attribution window (the time period within which the AI's contribution to the outcome can be claimed).

Example of a well-defined outcome definition: "Customer service ticket closed by the AI without human escalation (status = 'resolved', escalated = false in the customer service platform) within 24 hours of creation, with customer satisfaction score ≥ 4.0 as recorded in the post-resolution survey." This definition specifies the metric (ticket closure, no escalation, CSAT ≥ 4.0), the threshold (exact values specified), the measurement source (customer service platform and survey tool), and the attribution window (implied: the ticket must be created and closed within the billing period, or a specific attribution window must be defined).

Example of a poorly-defined outcome definition: "Successfully resolved customer service interactions." This definition has no metric, no threshold, no measurement source, and no attribution window. Every dispute about what "successfully resolved" means will be irresolvable because the contract does not define it.

The outcome definition belongs in the Price object's outcome_definition field and in the Outcome object's outcome_definition_id field. Both must reference the same precise definition, and that definition must be contractually agreed before any billing for outcomes begins.

Outcome Definition — Four Components			
Component	**What it specifies**	**Example (legal AI)**	**Common failure mode**

Metric	The specific number or state change that defines the outcome	Contracts reviewed with all material issues flagged (issue_flagged = true for any issue with severity ≥ 'medium')	'High-quality review' — not a metric. Must be a specific measurable result.
Threshold	The value the metric must reach to constitute a billable outcome	All issues with severity ≥ medium flagged within 4 hours of submission	'Good enough' — not a threshold. Must be a specific value both parties can compute.
Measurement source	Which system measures the metric and who controls it	Contract management system (customer-hosted); issue severity assigned by customer's legal team	Vendor's AI system is the source — creates conflict of interest and dispute risk
Attribution window	The time period in which the AI's contribution can be claimed	Contract submitted and review completed within the same billing month	No window defined — disputes about which period captures the outcome value

Outcome verification is the process of confirming that a defined business outcome has occurred. It is the event that triggers billing in an outcome-based commercial model, and it must be designed to be credible to both parties.

The ideal verification architecture has three properties. It is based on data from a system that neither party controls unilaterally — typically the customer's system of record. It is automated — the verification is triggered by the outcome event in the customer's system rather than by a manual claim from either party. And it is auditable — the verification event creates a record that both parties can query independently.

In practice, achieving all three properties simultaneously is challenging. The most common verification architecture is a webhook integration: the customer's system of record (CRM, ticketing system, contract management platform) is configured to send a webhook event when a defined outcome occurs. The vendor's billing system receives the webhook, validates that the outcome meets the definition criteria, creates an Outcome object with status = verified, and triggers a billing event. This architecture is partially automated (the webhook fires without human intervention) and based on customer

system data (the webhook comes from the customer's system), but it is not fully auditable without additional logging.

A more robust verification architecture uses a shared outcome ledger: a data store that both parties can write to and read from, with cryptographic guarantees that entries cannot be modified after creation. The customer's system writes an outcome record when the event occurs; the vendor's billing system reads the record and creates the billing event; both parties can audit the ledger independently. This architecture is more complex to implement but significantly more resistant to dispute.

The Outcome object's verified_by field records who or what verified the outcome. For automated verification from a customer system webhook, this field records the customer's system identifier. For manual verification (a human reviewing an outcome claim), it records the reviewer's ID. For third-party verification (an independent system assessing whether the SLA was met), it records the third-party system's identifier. The verified_by field is the attribution anchor for any dispute about whether an outcome was correctly verified.

Outcome Object — Complete Field Reference		
Field	**Type**	**Description / Values**
`id`	UUID v4 · required	Globally unique outcome record identifier.
`customer_id`	UUID · required	The customer for whom the outcome was delivered.
`product_id`	UUID · required	The outcome_service product that produced this outcome.
`outcome_definition_id`	UUID · required	References the specific outcome definition in the Price object. The adjudication standard.
`status`	enum · required	pending_verification | verified | rejected | disputed
`claimed_at`	ISO 8601 · required	When the outcome claim was first recorded. Starts the verification SLA clock.
`verified_at`	ISO 8601 · optional	When verification was confirmed. Triggers billing event on verification.

verified_by	string · required when verified	System or person that confirmed the outcome. The attribution anchor. e.g. 'customer_crm_webhook', 'legal_team_reviewer_id'
economic_value	decimal · optional	Estimated dollar value of this outcome to the customer. For gain-share calculations and ROI reporting.
billing_event_id	UUID · optional	The billing event triggered by this verified outcome. Confirms billing was generated.
rejection_reason	string · optional	If status = rejected: why the outcome was not verified. Triggers remediation workflow.
dispute_id	UUID · optional	If status = disputed: the associated billing dispute record.

Variable Consideration Under ASC 606

Outcome-based pricing creates variable consideration under ASC 606 and IFRS 15 — the portion of the transaction price that depends on performance and cannot be determined with certainty at contract inception. Variable consideration must be estimated, constrained, and disclosed in a way that produces reliable revenue recognition.

The constraint analysis for variable consideration in outcome-based AI pricing requires consideration of four factors. AI performance reliability: how consistently does the AI deliver outcomes at the contracted rate? For a newly deployed AI in a new use case, performance is uncertain, and the constraint should be conservative. Attribution disputes: how likely are disagreements about whether the AI caused the outcome? Higher dispute likelihood requires a more conservative variable consideration estimate. Measurement reliability: how accurate is the outcome measurement infrastructure? Measurement errors that systematically over- or under-count outcomes require adjustment in the estimate. Contract modification risk: how likely are mid-term changes to the outcome definition or measurement methodology? More likely modifications require more conservative estimates.

The practical implication for the pricing architect: outcome-based pricing contracts must include a variable consideration estimate methodology that the finance team has

reviewed and approved before the contract is signed. This is not a revenue recognition afterthought — it is a commercial design requirement. The outcome definition, the measurement methodology, and the attribution framework must all be designed with ASC 606 compliance in mind from the start.

Variable Consideration — Constraint Analysis Framework

Factor	Low constraint (recognize more)	High constraint (recognize less)	Assessment question
AI performance reliability	≥ 12 months of outcome delivery data; variance < 10%	New deployment; no historical data; high variance	How many months of outcome data do we have, and what is the variance?
Attribution dispute likelihood	Clear causal link; customer agrees AI caused outcomes	Multiple contributing factors; contested causation	What is our dispute rate on similar outcome claims?
Measurement reliability	Automated, customer-system measurement; audited	Manual verification; vendor-controlled measurement	What is the error rate in our outcome measurement?
Contract modification risk	Stable, well-defined outcomes; established relationship	Novel outcome type; new customer; ambiguous definition	How precise is the outcome definition and how mature is the relationship?

Chapter Ten — The Essentials

› Outcome definitions must specify metric, threshold, measurement source, and attribution window — all four, precisely.

› Verification sourced from the customer's system of record is more defensible than verification from the vendor's AI system.

› The Outcome object's verified_by field is the attribution anchor — it must record the specific system or person that verified the outcome.

› ASC 606 variable consideration analysis must be part of outcome pricing contract design — not a post-hoc accounting exercise.

› The constraint analysis framework determines how much variable consideration can be recognized — conservative estimates protect against restatement.

PART THREE

Offer Design and Catalog

From well-defined objects to commercial products customers can buy, quote, and contract.

CHAPTER ELEVEN

Idea to Offer: AI Product Catalog and Offer Design

Catalog architecture, good-better-best tiering for AI products, composite offer design, and lifecycle management.

The product catalog is where commercial intent becomes commercial reality. It is the structured data representation of everything the company offers for sale: every product, every price, every bundle, every tier, every promotional offer. The quality of the catalog determines the quality of everything downstream: the accuracy of quotes, the correctness of contracts, the reliability of billing, and the auditability of revenue recognition.

Most companies do not have a product catalog in the precise sense described here. They have a price list — a document that sales reps consult when creating proposals. The difference between a price list and a product catalog is not cosmetic. A price list is a document. A catalog is a data structure. A price list is consulted by humans. A catalog is consumed by systems — the CPQ system, the billing engine, the revenue recognition module. A price list has no lifecycle management. A catalog has versioning, activation states, and deprecation workflows. A price list accumulates inconsistencies over time. A catalog enforces consistency through schema validation.

Building a proper product catalog is one of the highest-leverage investments a RevOps or product team can make. It pays dividends in quote accuracy, billing reliability, and revenue recognition confidence every day it is in use.

Catalog Architecture: Three Levels

The AI product catalog has three levels of structure: the product tier (what is being sold), the offer tier (how it is packaged for sale to a specific customer segment), and the catalog tier (how products and offers are organized for discovery and selection).

At the product tier, each product in the catalog references the Product object definition from the data model: its type, its AI layer designation, its capability description, and its relationship to other products. A well-designed product tier has clear boundaries between products: each product represents a distinct commercial capability with its own entitlement and pricing logic. Products that differ only in configuration (different token limits, different model versions) should be represented as configurations of a base product, not as separate products, to avoid catalog proliferation.

At the offer tier, each offer combines a product with a price and a set of commercial terms appropriate for a specific customer segment. An offer for the self-service market might combine a base product with a freemium price structure, a monthly subscription floor, and a consumption overage component, packaged for easy online purchase without sales involvement. An offer for the enterprise market might combine the same base product with a negotiated enterprise price, an annual commitment, a custom SLA, and a consumption commitment that activates specific volume discounts.

At the catalog tier, products and offers are organized into a browsable, searchable structure that allows buyers (human or AI agent) to find the right product for their needs. Catalog organization for AI products requires attention to the discovery vocabulary: the tags, the capability descriptions, and the use case mappings that allow a buyer who knows what they need to find the product that provides it. This is particularly

important for the agent commerce channel, where AI agents will be searching the catalog programmatically based on capability tags rather than browsing by category.

Catalog Architecture — Three Levels

Level	What it contains	Managed by	Consumed by
Product tier	Product objects: type, ai_layer, capability description, version	Product management + catalog governance	All downstream levels; CPQ; agent discovery
Offer tier	Product + Price + Entitlement structure + commercial terms for a specific segment	Pricing strategy + RevOps	CPQ; contract management; billing engine
Catalog tier	Organized, tagged, searchable presentation of offers by segment, use case, layer	Marketing + RevOps	Customer portal; sales tooling; agent catalog search

Good-Better-Best Tiering for AI Products

The good-better-best tiering structure — three or four product tiers that scale capability and price — is the dominant packaging pattern for AI application products, and for good reason. It is familiar to buyers, it creates natural upgrade paths, and it allows the vendor to serve multiple customer segments with a single product.

Designing the good-better-best structure for an AI product requires more care than for a SaaS product, because the consumption dimensions are more complex. A SaaS good-better-best structure typically differentiates on features: the higher tier includes more features than the lower tier. An AI good-better-best structure must differentiate on multiple dimensions simultaneously: feature access, consumption volume, AI capability quality (which model powers the tier), and service level (response time, uptime commitment).

The token allocation per tier deserves particular attention. The allocation should be set based on actual consumption data from existing deployments, not based on intuition or competitive benchmarking. An allocation that is too low causes customers to exhaust their budget mid-period and either interrupt their workflows or face expensive overage charges. An allocation that is too high means customers are paying for capacity they never use — which is wasteful for them and reduces the natural pressure to upgrade.

The model quality differentiation between tiers — the Standard tier uses a smaller, faster, cheaper model; the Professional tier uses a larger, more capable, more expensive model — is a powerful differentiation mechanism that directly aligns price with value. The challenge is that model quality differentiation creates different underlying cost structures for different tiers, which must be reflected in the Meter configuration (different token cost multipliers for different models) and in the margin analysis for each tier.

Good-Better-Best Tier Design Framework

Dimension	Standard tier	Professional tier	Enterprise tier	Design principle
Foundation model	Smaller, faster model (e.g., Haiku-class)	Balanced model (e.g., Sonnet-class)	Largest, most capable model (e.g., Opus-class)	Quality differentiation must be real and perceivable
Token allocation	10M tokens/month	50M tokens/month	500M tokens/month	Set at p50 of expected consumption per segment
Overage pricing	$0.008/1K tokens	$0.006/1K tokens	$0.004/1K tokens	Lower overage rate rewards higher-tier commitment
Rate limit	60 requests/min	300 requests/min	Unlimited (governed by SLA)	Rate limits should reflect infrastructure cost, not be arbitrary

SLA	99% availability · 5s P99 latency	99.5% availability · 2s P99 latency	99.9% availability · 500ms P99 latency	SLA differentiation should reflect actual cost-to-serve differences
Support	Self-service + docs	Priority email + 8h response	Dedicated CSM + 1h response	Support cost must be priced into the tier margin
Pricing model	Subscription only	Subscription + task overage	Outcome-based option available	Higher tiers unlock higher-value pricing models

FOR THE PRICING STRATEGIST

Token allocation sizing is the most important and least rigorous tier design decision

Most good-better-best tier designs set token allocations based on competitive benchmarking (what does the competitor charge?) or intuition (what sounds right?) rather than on actual consumption data. This produces allocations that are systematically wrong: too low for the target segment (causing budget exhaustion and frustration) or too high (meaning customers never feel pressure to upgrade). Before finalizing tier allocations, run consumption modeling against actual usage data from existing customers, segmented by customer size and use case. Set the Standard allocation at p50 consumption for the target segment, Professional at p75, and Enterprise at p95. Review and adjust annually as consumption patterns evolve.

Offer Lifecycle Management

AI product offers have shorter commercial lifespans than SaaS offers because the underlying technology changes faster. An offer designed in 2024 for a specific model capability may be obsolete by 2026 as model capabilities improve dramatically. The catalog must be designed with lifecycle management in mind from the start.

Three lifecycle events require explicit management. Offer activation: the process by which a draft offer becomes available for sale, including the approval workflow that ensures the offer is internally consistent and commercially sound before it reaches customers. Offer deprecation: the process by which an existing offer is retired — stopping new sales while honoring existing contracts for their remaining term. Offer migration: the process by which existing customers on a deprecated offer are

transitioned to a current offer — the most operationally sensitive lifecycle event because it changes the terms of an existing commercial relationship.

The deprecation and migration workflows are the ones most companies are least prepared for, because they require coordinating changes across the product catalog, the billing system, the contract management system, and the customer success motion simultaneously. Building these workflows explicitly — with defined trigger criteria, defined notice periods, defined migration terms, and defined escalation paths for customers who object — is one of the most valuable investments a RevOps team can make in catalog governance.

Chapter Eleven — The Essentials
› Catalog architecture has three levels: product tier (what), offer tier (how and for whom), catalog tier (organized for discovery).
› Good-better-best tiers for AI products differentiate on model quality, token allocation, overage rates, rate limits, SLA, and support.
› Token allocation sizing must be based on actual consumption data, not competitive benchmarking or intuition.
› Offer lifecycle management — activation, deprecation, migration — must be designed as explicit workflows, not improvised when needed.
› Higher tiers should unlock higher-value pricing models (outcome-based) in addition to higher consumption limits.

CHAPTER TWELVE

CPQ for AI: Configure, Price, Quote at Speed

Consumption modeling, quote approval workflows, dynamic pricing, and AI-native CPQ architecture.

Configure, Price, Quote — CPQ — is the process by which a product catalog is translated into a specific commercial proposal for a specific customer. For AI products, CPQ is significantly more complex than for SaaS products because AI proposals must accurately represent variable pricing components, consumption estimates, outcome definitions, and SLA commitments that do not exist in the SaaS CPQ playbook.

An AI-native CPQ capability has three components that traditional CPQ lacks: consumption modeling (the ability to estimate a customer's likely consumption based on their use case and scale, which determines the variable billing components of the proposal), outcome definition tooling (guided workflows for specifying and agreeing on outcome definitions that will become the basis for outcome-based billing), and SLA architecture (the ability to configure AI-specific SLA terms — quality, latency, availability — and model their financial implications in the proposal).

Consumption Modeling

Consumption modeling for CPQ is the practice of estimating how much of each pricing dimension a specific customer will likely consume in a given period. For token-based products, this means estimating the number of input and output tokens generated per use case per user per month. For task-based products, this means estimating the number of task completions per team per month. For outcome-based products, this means estimating the number of verifiable outcomes per deployment per month.

Consumption estimates affect the proposal in two ways. They determine the appropriate tier or base commitment level — a customer with estimated consumption of 50 million tokens per month should not be proposed a tier with a 10 million token allocation. And they determine the projected variable billing — the expected overage charges that will appear on future invoices if the customer's consumption exceeds their base commitment.

Accurate consumption estimates are the most direct lever for preventing first-invoice surprises. A proposal that shows a customer their expected total cost of ownership —

base commitment plus projected overages at the estimated consumption rate — is a proposal that is unlikely to generate a billing dispute when the first invoice matches the estimate. A proposal that shows only the base commitment, with the overage pricing buried in the terms, is a proposal that will generate a surprised and unhappy customer when their first invoice is significantly higher than expected.

The CPQ system should maintain a consumption modeling library: aggregated, anonymized consumption data from existing customers by use case and scale, which can be used to model expected consumption for prospects with similar profiles. This library is built from the Event data in the metering system — another reason why investing in event-granularity metering pays dividends beyond billing accuracy.

Consumption Modeling — Input Variables and Estimation Methods

Variable	What it drives	Estimation method	Data source
Monthly interaction volume	Total token consumption for interactive features	Interviews + benchmark data from similar deployments	CSM notes + existing customer event data
Average tokens per interaction	Per-interaction token cost	Prompt engineering analysis + model API test calls	Product team + ML team
Agent workflow frequency	Total task completion volume	Process analysis: how many times per month does the customer run this workflow?	Customer workshop + existing customer task data
Average tokens per task	Per-task token cost	Workflow test runs across representative inputs	ML Ops team
Document volume and size	Token consumption for document-processing features	Customer's current document processing metrics	Customer's ops team in discovery
Growth rate	Consumption in months 2–12	Historical growth rate for similar deployments	Customer cohort analysis

Quote approval workflows for AI products must handle two dimensions of complexity that SaaS approval workflows do not: variable component risk and SLA commitment risk.

Variable component risk is the exposure created by consumption-based pricing terms. When a sales rep proposes a token allocation that is substantially below the likely consumption level — either to make the initial price look attractive or to minimize the customer's perceived financial commitment — the revenue from the expected overages is uncertain, the customer's experience when the overages materialize may be negative, and the overall deal economics may be worse than a properly structured deal with a higher base commitment. The approval workflow should include a review of the consumption estimate and the projected overage revenue, not just the base commitment.

SLA commitment risk is the exposure created by performance commitments that may be difficult to meet. An AI product that commits to a 95% task completion rate on a use case where the model currently achieves 90% is committing to a service level it cannot reliably deliver. When the SLA is breached, the customer receives credits that reduce revenue. When the SLA breach is significant, the customer may have grounds to terminate the contract. The approval workflow should include a review of the proposed SLA against the AI's historical performance on comparable use cases.

The deal desk chapter in the leaders' volume described the six components of a well-constructed AI deal. The CPQ approval workflow is the operational implementation of that structure: a systematic check that each component — base commitment, variable pricing, SLA, price adjustment, IP governance, expansion mechanism — is present, correctly configured, and commercially sound before the quote is sent to the customer.

CPQ Approval Workflow — Trigger Criteria for AI Deals

Review trigger	Who reviews	What they assess	Time limit
Discount > 20% of list price	Sales director + RevOps	Margin impact; competitive intelligence; precedent risk	24 hours

Consumption estimate below p25 for segment	RevOps + CSM	Risk of first-invoice surprise; appropriate tier for stated use case	Same day
Custom outcome definition (not standard)	Product + Legal + RevOps	Measurability; attribution; ASC 606 variable consideration impact	48 hours
SLA commitment above current performance data	Product + ML Ops	AI performance vs committed SLA; breach credit exposure	48 hours
Contract value > $500K	VP Sales + CFO	Overall deal economics; strategic value; resource commitment	72 hours
Non-standard IP or data governance terms	Legal	IP ownership; model training rights; data sovereignty	72 hours

AI-Native CPQ Architecture

Dynamic pricing — adjusting prices in real time based on demand, availability, or customer behavior — is nascent at the AI application layer but standard at the compute layer and increasingly available at the model layer.

At the compute layer, spot pricing is the primary form of dynamic pricing: prices fluctuate based on current demand for GPU capacity, with prices falling when utilization is below target and rising when demand exceeds supply. Implementing spot pricing requires a pricing API that quotes current prices in real time, a billing system that records the price at the time of consumption, and a reconciliation capability that handles the discrepancy between quoted prices and actual billed prices when prices change mid-session.

At the model layer, dynamic pricing based on request load is less common but emerging: some API providers vary prices based on time of day or current system load, with lower prices during off-peak hours. Implementing this requires the same real-time price API and historical price recording as compute spot pricing.

At the application layer, dynamic pricing based on customer behavior — adjusting prices based on utilization rates, value delivered, or competitive threat — is theoretically

possible but operationally complex. The primary risk is customer trust damage: customers who discover that the same capability is being priced differently for different customers based on perceived willingness to pay are likely to feel exploited. Dynamic pricing at the application layer works best when the variation is transparent — explicitly defined tiers with different characteristics — rather than when it is opaque.

An AI-native CPQ system has three capabilities beyond traditional CPQ that are specifically required for AI product quoting. The first is the consumption modeling library — a structured database of anonymized consumption patterns by product type, use case, and customer segment, used to generate consumption estimates for prospects. The second is the outcome value calculator — a tool that estimates the economic value of AI-delivered outcomes for a specific customer, enabling value-based pricing conversations grounded in customer-specific evidence. The third is the SLA risk modeler — a tool that assesses the AI's historical performance against the SLA terms being proposed and calculates the expected credit exposure from SLA breaches under different performance scenarios.

These three capabilities transform the CPQ process from a price-lookup exercise into a value-modeling exercise. The sales team is no longer looking up the standard price and applying a standard discount. They are modeling the customer's expected consumption, estimating the value the AI will create, designing SLA terms that are appropriately calibrated to the AI's actual performance, and building a proposal that the customer's CFO will find credible because it is grounded in data rather than assertion.

> **FOR THE REVOPS LEAD**
>
> **The consumption modeling library pays for itself on the first prevented dispute**
>
> The cost of building a consumption modeling library — anonymizing and segmenting existing customer event data, building the estimation tool, training the sales team to use it — is typically $50–100K in RevOps and engineering time. The cost of a first-invoice dispute on a major enterprise account — customer calls, credit memos, relationship damage, potential churn — often runs to the same order of magnitude, plus the senior time invested in resolution. Build the library before you have your first enterprise consumption billing dispute, not after.

Chapter Twelve — The Essentials

› Consumption modeling transforms CPQ from a price-lookup exercise into a value-modeling exercise — preventing first-invoice surprises.

› AI CPQ approval workflows must address four AI-specific risk dimensions: consumption estimate accuracy, outcome definition quality, SLA calibration, and IP terms.

› The consumption modeling library should be built from existing customer event data — anonymized and segmented by use case and customer profile.

› Dynamic pricing at the application layer works best when variation is transparent (defined tiers) rather than opaque (willingness-to-pay pricing).

› AI-native CPQ requires three capabilities: consumption modeling, outcome value calculation, and SLA risk modeling.

CHAPTER THIRTEEN

Pricing Strategy by Layer: The Practitioner's Matrix

Decision trees for pricing architects. Matching model to layer to maturity to implementation.

The practitioner's pricing strategy matrix combines the five-layer AI economy framework with the ten monetization models to produce a specific, actionable pricing recommendation for any given company position and product type.

The matrix is organized in two dimensions: the layer of the AI economy at which the product primarily operates (columns) and the company's commercial maturity stage (rows). The intersection of layer and maturity stage determines the recommended primary pricing model, the recommended secondary pricing model (if applicable), and the key implementation prerequisites.

For an early-stage AI application company operating at the agent layer, the matrix recommends: primary model = subscription with usage metering (to acquire customers

and collect consumption data), secondary model = per-task (introduce as an add-on for high-volume customers), prerequisites = task definition precision, event metering for task completions, billing system capable of handling task-based billing alongside subscription billing.

For a growth-stage AI application company operating at the outcome layer, the matrix recommends: primary model = hybrid subscription floor plus outcome-based component, secondary model = gain-share for strategic accounts, prerequisites = outcome definition agreed with at least 10% of the customer base, measurement infrastructure for at least two major outcome types, ASC 606 variable consideration policy reviewed by finance.

For a mature AI application company with proven outcome delivery, the matrix recommends: primary model = outcome-based with optional subscription floor, secondary model = gain-share for enterprise strategic accounts, prerequisites = all the above, plus automated outcome verification for the majority of outcomes billed, BHI ≥ 99% for billing accuracy.

The matrix is a guide, not a mandate. The specific pricing decisions for any company depend on factors the matrix cannot fully capture: competitive dynamics, customer sophistication, technical capability, and the specific nature of the AI capability being priced.

The Practitioner's Pricing Strategy Matrix

Company type	AI layer	Early stage	Growth stage	Mature stage
Infra provider	Compute	On-demand + spot · establish utilization data	Reserved + on-demand · calibrate reservation economics	Hybrid reserved+spot · enterprise contracts · AI factory pricing
Model company	Model	Per-token basic (input/output) · acquire developer base	Per-token quality tiers · subscription floor for enterprise	Per-token + fine-tune royalties + marketplace distribution

AI application	Agent	Subscription + usage metering (data collection phase)	Per-task + subscription floor · introduce outcome pilots	Outcome-based primary · gain-share for strategic accounts
AI application	Outcome	Subscription + outcome reporting (pre-billing)	Hybrid subscription + outcome component · limited use cases	Pure outcome + optional subscription floor · full outcome catalog
Enterprise buyer	Internal FinOps	Showback (visibility without chargeback)	Chargeback by team · token budget governance	Full FinOps OS: chargeback + optimization + governance + audit

The Model Selection Decision Tree

The matrix above provides the directional recommendation. The decision tree below provides the implementation logic — the specific questions a pricing architect must answer to move from the matrix recommendation to the final pricing model selection.

Pricing Model Selection Decision Tree	
Is outcome measurement infrastructure in place? (Can you measure and attribute outcomes reliably?)	YES → outcome-based pricing is available. Proceed to attribution question. NO → do not attempt outcome pricing. Use per-task or subscription while building measurement infrastructure.
Is attribution methodology agreed with customers? (Will customers accept your attribution logic?)	YES → proceed to accounting question. NO → outcome pricing is available internally but not commercially yet. Use per-task pricing with outcome reporting.
Has the variable consideration methodology been reviewed by finance?	YES → outcome-based pricing can be implemented commercially. NO → do not sign outcome-based contracts until finance completes the ASC 606 analysis.
Are task definitions precise enough to adjudicate disputes without ambiguity?	YES → per-task pricing is available. NO → use subscription or hybrid subscription+task until task definitions are refined.

Is the metering infrastructure capable of event-level tracking at expected volume?	YES → consumption-based billing (per-token or per-task) is operationally viable. NO → use subscription until metering infrastructure is in place. Do not launch consumption billing on inadequate metering.
Are customers technically sophisticated enough to manage consumption-based billing?	YES → per-token or per-task pricing is appropriate. NO → use subscription with consumption allocation (tier model) to abstract billing complexity.

Chapter Thirteen — The Essentials

› The pricing strategy matrix maps company type × AI layer × maturity stage to specific pricing model recommendations.

› The decision tree operationalizes the matrix — six questions that determine which pricing model is both desirable and operationally viable.

› Never launch outcome pricing without: measurement infrastructure, agreed attribution methodology, and finance sign-off on variable consideration treatment.

› Never launch consumption billing without metering infrastructure capable of event-level tracking at expected volume.

› The right pricing model for today is not the right pricing model forever — build the prerequisites for the next stage while operating the current one.

CHAPTER FOURTEEN

The Monetization Maturity Ladder: Implementation Guide

How to execute the seat-to-usage-to-outcome migration. Triggers, prerequisites, risks, and the five-phase playbook.

The migration from one pricing model to the next — moving up the monetization maturity ladder — is the most commercially sensitive operation in AI product management. Done well, it increases revenue, strengthens customer relationships, and builds the commercial infrastructure for long-term growth. Done poorly, it generates customer anger, billing disputes, and revenue attrition.

The migration playbook has five phases that must be executed in sequence.

Phase one is data collection: one to three months of instrumented consumption measurement at the granularity required for the new pricing model. If the destination is per-task pricing, the instrumentation must capture task completion events. If the destination is outcome pricing, the instrumentation must capture outcome verification events. This phase generates no billing changes — it is purely about building the evidence base.

Phase two is customer communication: presenting the consumption data to customers and having honest conversations about what the data reveals about usage patterns and value delivery. Customers who have been shown their consumption data — how many tasks were completed, what outcomes were delivered, what those outcomes were worth to them — are much more receptive to pricing changes than customers who receive a pricing change announcement without context.

Phase three is model design: finalizing the new pricing model based on the consumption data and customer feedback from phase two. This includes setting the price per unit, designing the tier structure, and determining the transition terms for existing customers.

Phase four is piloting: introducing the new model with new customers and with the segment of existing customers that has the most to gain from it (typically the heaviest users, who are currently under-priced under the old model).

Phase five is migration: moving the installed base to the new model at renewal, with sufficient notice, clear explanation of the new terms, and support for customers who need help modeling their expected costs under the new structure.

> ***"The migration from one pricing model to the next is the most commercially sensitive operation in AI product***

> ***management. Done well, it builds trust and revenue. Done poorly, it generates anger, disputes, and attrition."***

The Five-Phase Migration Playbook

Step 1: Data Collection

Instrument the product at the granularity required for the new pricing model. For per-task migration: task completion events must be live and validated. For outcome migration: outcome verification events must be live and validated. Run for a minimum of 90 days to build a reliable consumption distribution. Produce a consumption analysis showing p50, p75, p90, and p95 consumption per customer segment — this data drives every subsequent decision.

Step 2: Customer Communication

Share the consumption data with customers in a customer-friendly format. Show each customer their specific consumption pattern and what it would cost under the new pricing model at different price levels. This conversation does three things: it builds credibility (you know their usage better than they do), it surfaces objections before they become disputes, and it identifies the customers who will welcome the new model versus those who will resist it.

Step 3: Model Design

Finalize the pricing model parameters based on consumption data and customer feedback. Set per-unit prices at the level that generates target economics for the median customer. Design the transition terms for existing customers: typically a grandfather period of 6–12 months at current pricing, followed by migration to new pricing at renewal. Design the migration incentive: customers who opt in to new pricing early receive a discount or additional capability that makes early adoption attractive.

Step 4: Pilot

Launch the new model with three to five customers: ideally a mix of a new customer, a heavy user who is currently under-priced, and a light user who needs reassurance that the new model is fair for them. Run the pilot for a full billing cycle. Review the billing accuracy, the customer response, the dispute rate, and the operational burden. Fix any issues found in the pilot before scaling.

Step 5: Migration

Move the installed base to the new model in three waves: willing adopters first (customers who expressed interest in the new model during phase 2), at-risk customers second (heavy users who are currently under-priced and have the most to lose from a delayed transition), legacy customers last (light users who are comfortable with the current model and will need the most support through the transition). Provide dedicated CSM support for the at-risk wave — these are the customers most likely to churn if the transition is handled poorly.

Migration Risk Mitigation — By Customer Segment

Customer segment	Migration risk	Mitigation approach	Success metric
Heavy users (top 20% by consumption)	May resist paying more even when value is clear	Run outcome ROI analysis showing value delivered; design migration terms that feel proportional to value received	Migration completed without NRR impact > 5%
Light users (bottom 30% by consumption)	May perceive consumption pricing as unfair since they use little	Set subscription floor that protects light users; communicate explicitly that the floor prevents overpayment	No churn spike in light-user segment at first renewal post-migration
Technical buyers	May want to optimize against the new pricing unit aggressively	Design the pricing model so that optimization benefits the customer without destroying the vendor's economics	Consumption optimization does not reduce margin below acceptable floor

Finance-controlled buyers	Need to preserve budget predictability	Include subscription floor in all enterprise migration offers; provide 12-month consumption forecast at migration	90% of enterprise customers accept the subscription floor structure
Strategic / logo accounts	High relationship sensitivity; any friction is visible	White-glove migration support; executive sponsor involvement; customized migration terms if needed	Zero strategic account churn attributable to pricing migration

Chapter Fourteen — The Essentials

› Five-phase migration: data collection (90+ days), customer communication, model design, pilot (one full billing cycle), full migration.

› Segment the migration: willing adopters first, at-risk customers second, legacy customers last — with different support levels for each.

› The subscription floor is the universal migration anxiety reducer — use it for every enterprise customer in the migration.

› Run the pilot for a full billing cycle before scaling — billing issues that don't appear in the first week will appear in the first month.

› Never migrate a strategic account without executive sponsor involvement — relationship risk is higher than economics risk for logo customers.

CLOSING

Build the Foundation. Then Build on It.

The architecture decisions made here compound over time. Make them deliberately.

The work described in this book is unglamorous. Defining thirteen objects with precise schemas does not ship a product. Building a metering infrastructure that captures events at sub-second granularity does not close a deal. Designing a consumption modeling library does not win a competitive evaluation. None of this work is visible to customers or to investors in the quarter it is done.

But it is the work that determines whether the commercial architecture can support the business the AI products are building. A metering infrastructure that loses 3% of events costs the same to build as one that loses 0% — but the 3% loss accumulates as revenue leakage that grows with scale. A product catalog that cannot represent composite pricing requires the billing team to handle composite deals manually — which scales until it does not. A token budget governance system that does not exist allows AI spending to grow unconstrained until the CFO stops it — usually at a moment that disrupts active deployments.

The foundation is not the most exciting part of the architecture. It is the part that makes everything else work. Build it with precision. Build it before you need it. And build it in the order this book recommends: data model first, pricing layer implementation second, offer design and catalog third.

The companies that do this work early and do it well will have commercial architectures that compound their AI product advantages over time. The companies that defer it will find that their commercial infrastructure is the constraint on their growth — the thing that prevents them from capturing the value their AI creates at the scale their AI can deliver.

> ***"Get the data model right before you build anything else. Pricing strategy is architecture. And architecture, done right, compounds."***

The AI Economy Monetization Series continues in Book Two-B:

The Commercial Pipeline and A/R