

Subscription & Billing System: Design Document

Overview

A comprehensive subscription management platform that handles recurring billing, usage tracking, and customer lifecycle management. The key architectural challenge is maintaining consistency across distributed billing operations while supporting complex pricing models, proration calculations, and real-time usage metering.

This guide is meant to help you understand the big picture before diving into each milestone. Refer back to it whenever you need context on how components connect.

Context and Problem Statement

Milestone(s): Foundation for all milestones - establishes the business and technical context

Mental Model: The Digital Subscription Economy

Think of subscription billing like running a **premium gym with multiple membership tiers**. In the physical world, a gym might offer basic access, premium classes, and VIP concierge services at different monthly rates. Members can upgrade from basic to premium mid-month (paying the difference), pause their membership during vacation, or cancel at the end of their billing cycle. The gym needs to track who has access to which facilities, when payments are due, how much to charge for partial months, and what happens when a member's credit card expires.

Now imagine this gym operates across multiple time zones, serves millions of members simultaneously, never closes for maintenance, and offers hundreds of different membership combinations. The digital subscription economy amplifies every complexity of physical membership management by orders of magnitude.

Subscription billing systems are the digital equivalent of that gym's membership office, but they must handle several additional layers of complexity that don't exist in physical businesses:

- 1. Instant Global Scale:** A software product can acquire thousands of customers overnight across different countries, currencies, and regulatory environments. Unlike a physical gym that grows gradually, digital products can experience explosive growth that breaks traditional billing approaches.
- 2. Complex Usage Patterns:** Digital services often combine fixed monthly fees with variable usage charges. Think of it like a gym membership that includes basic access but charges extra per personal training session, except the "sessions" might be API calls, storage gigabytes, or processed transactions measured in real-time.

3. Continuous Service Delivery: Physical gyms can temporarily restrict access for non-payment, but digital services face the challenge of real-time entitlement decisions. When a customer's payment fails, the system must instantly determine what features to disable while maintaining a path to service restoration.

4. Microscopic Financial Precision: Digital billing operates at scales where rounding errors of a few cents, multiplied across millions of customers and thousands of plan changes, can result in significant revenue leakage or compliance violations.

The mental model throughout this design will be that we're building **a fully automated membership management system** for a service that never sleeps, serves global customers, and must make split-second decisions about service access based on complex financial calculations.

Existing Approaches and Trade-offs

When building subscription billing capabilities, engineering teams face a fundamental **build-versus-buy decision** that significantly impacts development velocity, operational complexity, and long-term business flexibility. The landscape includes several mature third-party solutions alongside the option of building a custom system.

Third-Party Billing Platforms

Stripe Billing represents the most popular third-party approach, offering a comprehensive subscription management layer on top of Stripe's payment processing infrastructure. The platform provides pre-built subscription lifecycle management, automatic proration calculations, and extensive webhook integration for real-time billing events.

Capability	Stripe Billing	Custom Solution
Time to Market	2-4 weeks integration	6-12 months development
Proration Logic	Pre-built, battle-tested	Must implement and debug
Payment Failures	Automatic retry policies	Custom dunning management
Tax Compliance	Built-in tax calculation	Manual tax service integration
Multi-currency	Native support	Exchange rate management needed
Usage Metering	Basic usage reporting	Full control over aggregation
Complex Pricing	Limited to Stripe's models	Unlimited customization
Vendor Lock-in	High - data and logic coupling	None - full ownership
Monthly Cost	\$0.5% of revenue + payment fees	Infrastructure + development costs

Chargebee and **Recurly** offer similar comprehensive billing platforms with slightly different feature emphases. Chargebee excels at complex pricing models and revenue recognition, while Recurly focuses on subscription

analytics and churn reduction features.

Specialized Platforms like Lago, Metronome, and Togai have emerged specifically for usage-based billing scenarios. These platforms excel at high-volume event ingestion and complex usage aggregation but may lack the subscription lifecycle sophistication of general-purpose billing platforms.

Custom Solution Trade-offs

Building a custom subscription billing system offers complete control over business logic and data ownership, but introduces significant engineering complexity and ongoing maintenance responsibilities.

Key Insight: The decision isn't just about initial development effort - it's about accepting long-term responsibility for financial accuracy, compliance, and operational reliability in a domain where mistakes directly impact revenue.

Advantages of Custom Development:

- **Business Logic Flexibility:** Complete control over pricing models, proration algorithms, and subscription lifecycle policies. This becomes crucial for businesses with unique pricing strategies or complex B2B arrangements that don't fit standard platform templates.
- **Data Ownership and Integration:** Direct access to all billing and usage data without API limitations. This enables sophisticated analytics, machine learning applications, and seamless integration with existing business systems.
- **Cost Predictability:** Fixed infrastructure costs rather than percentage-based fees that scale with revenue. For high-volume businesses, this can represent significant savings compared to platform fees.
- **Performance Optimization:** Ability to optimize database schemas, caching strategies, and processing pipelines for specific usage patterns and scale requirements.

Disadvantages of Custom Development:

- **Domain Complexity Underestimation:** Subscription billing appears straightforward but contains numerous edge cases around calendar calculations, timezone handling, currency precision, and regulatory compliance. Teams consistently underestimate the effort required to handle these correctly.
- **Financial Accuracy Responsibility:** Custom systems must achieve the same level of financial precision and audit compliance as established platforms, but without the benefit of years of production hardening and edge case discovery.
- **Ongoing Maintenance Burden:** Billing systems require continuous updates for payment processor changes, tax regulation updates, currency support, and security compliance requirements.
- **Expertise Requirements:** Building reliable billing systems requires deep understanding of payment processing, financial regulations, tax compliance, and accounting principles - expertise that may not exist within typical engineering teams.

Hybrid Approaches

Many successful implementations combine third-party platforms for core payment processing and basic subscription management with custom systems for business-specific logic and advanced analytics.

Payment Gateway + Custom Billing Logic: Use established payment processors (Stripe, Braintree) for payment collection and PCI compliance while building custom subscription management, proration, and usage tracking systems. This approach provides payment reliability while maintaining business logic control.

Platform Integration with Custom Extensions: Implement core billing through platforms like Stripe or Chargebee while building custom systems for specialized requirements like complex usage aggregation, custom pricing models, or advanced analytics that exceed platform capabilities.

Architecture Decision: Custom System with Payment Gateway Integration

- **Context:** This design document focuses on building a custom subscription billing system that integrates with an existing payment gateway (from the prerequisite payment-gateway project). This represents a middle-ground approach that maintains control over business logic while leveraging proven payment processing infrastructure.
- **Options Considered:**
 1. Full third-party platform (Stripe Billing)
 2. Hybrid payment platform with custom subscription logic
 3. Completely custom payment and billing system
- **Decision:** Custom billing system with payment gateway integration (Option 2)
- **Rationale:** Provides maximum business logic flexibility and data ownership while avoiding the complexity and compliance burden of payment processing. Enables sophisticated usage-based billing and custom proration logic that may not fit platform constraints.
- **Consequences:** Requires building and maintaining complex billing logic, but enables unlimited customization and direct data access. Development timeline extends 6-12 months compared to platform integration, but eliminates ongoing percentage-based fees and vendor lock-in.

Technical Challenges in Custom Billing Systems

Building reliable subscription billing systems presents several technical challenges that don't exist in typical CRUD applications:

Financial Precision and Consistency: Billing calculations must maintain perfect accuracy across concurrent operations, system failures, and data migrations. Unlike other domains where eventual consistency may be acceptable, financial systems require immediate consistency with full audit trails.

Complex State Management: Subscriptions, invoices, and payment states interact in intricate ways that create potential race conditions and inconsistent state scenarios. The system must handle concurrent plan changes, payment processing, and usage updates without compromising data integrity.

Time-Sensitive Processing: Billing cycles, trial expirations, and payment retries operate on strict time schedules that cannot be missed. The system must reliably process time-sensitive events even during high load periods or infrastructure failures.

Multi-Tenant Data Isolation: Billing systems often serve multiple business entities or customer segments with different pricing rules, tax requirements, and compliance needs. The architecture must ensure complete data isolation while maintaining operational efficiency.

Integration Complexity: Modern billing systems integrate with payment processors, tax calculation services, accounting systems, CRM platforms, and business intelligence tools. Each integration introduces potential failure modes and data synchronization challenges.

Regulatory Compliance: Subscription billing systems must comply with financial regulations, data privacy laws, and industry-specific requirements that vary by geography and business model. These requirements often change over time, requiring system flexibility.

Critical Design Principle: In billing systems, correctness is more important than performance, but the system must be designed to achieve both. Financial accuracy cannot be compromised for faster processing, but slow billing operations create poor customer experiences and operational challenges.

The remainder of this design document provides detailed guidance for building a custom subscription billing system that addresses these challenges while maintaining the flexibility advantages that justify custom development over third-party platforms.

Implementation Guidance

The implementation approach for this subscription billing system prioritizes **financial accuracy and auditability** over development speed. The technology choices reflect the need for strong consistency, precise decimal calculations, and reliable event processing.

Technology Recommendations

Component	Simple Option	Advanced Option
Database	PostgreSQL with ACID transactions	PostgreSQL + Redis for session data
Decimal Precision	Python <code>decimal.Decimal</code> library	Custom fixed-point arithmetic
Task Scheduling	APScheduler for billing cycles	Celery + Redis for distributed tasks
API Framework	FastAPI with Pydantic validation	Django REST Framework
Event Processing	Direct database triggers	Event sourcing with Kafka
Monitoring	Structured logging + Prometheus	Full observability stack (traces, metrics, logs)
Testing	pytest with factory-boy	Property-based testing with Hypothesis

Recommended Project Structure

The codebase organization separates business domain logic from infrastructure concerns, enabling easier testing and future scaling decisions:

```
subscription-billing/
├── src/billing/
│   ├── __init__.py
│   ├── domain/           # Core business logic
│   │   ├── __init__.py
│   │   ├── models/        # Business entities
│   │   │   ├── customer.py
│   │   │   ├── plan.py
│   │   │   ├── subscription.py
│   │   │   └── invoice.py
│   │   ├── services/      # Business operations
│   │   │   ├── plan_manager.py
│   │   │   ├── subscription_engine.py
│   │   │   ├── proration_calculator.py
│   │   │   └── usage_tracker.py
│   │   └── exceptions.py
│   ├── infrastructure/   # External integrations
│   │   ├── database/
│   │   │   ├── repositories.py
│   │   │   └── migrations/
│   │   ├── payment_gateway/
│   │   │   └── gateway_client.py
│   │   ├── events/
│   │   │   └── webhook_handlers.py
│   │   └── tasks/
│   │       └── billing_jobs.py
│   ├── api/              # HTTP interface
│   │   ├── __init__.py
│   │   ├── routers/
│   │   │   ├── plans.py
│   │   │   ├── subscriptions.py
│   │   │   └── usage.py
│   │   ├── schemas/        # Request/response models
│   │   │   └── dependencies.py
│   │   └── config/
│   │       ├── settings.py
│   │       └── logging.py
│   ├── tests/
│   │   ├── unit/
│   │   ├── integration/
│   │   └── fixtures/
│   ├── scripts/
│   │   ├── migrate_data.py
│   │   └── billing_reconciliation.py
│   └── docker-compose.yml
└── requirements.txt
└── README.md
```

Infrastructure Starter Code

Database Connection and Transaction Management (`infrastructure/database/connection.py`):

```
import contextlib

import logging

from decimal import Decimal

from typing import Generator

import psycopg2

from psycopg2.extras import RealDictCursor

from psycopg2.extensions import ISOLATION_LEVEL_SERIALIZABLE

logger = logging.getLogger(__name__)

class DatabaseManager:

    """Manages database connections with transaction support for billing operations."""

    def __init__(self, connection_string: str):
        self.connection_string = connection_string
        self._setup_decimal_adapter()

    def _setup_decimal_adapter(self):
        """Configure psycopg2 to handle Python Decimal objects properly."""
        from psycopg2.extensions import register_adapter
        from psycopg2 import sql

        def adapt_decimal(decimal_obj):
            return sql.Literal(str(decimal_obj))

        register_adapter(Decimal, adapt_decimal)

    @contextlib.contextmanager
```

```
def transaction(self, isolation_level=ISOLATION_LEVEL_SERIALIZABLE) -> Generator:

    """
    Provides transactional context for billing operations.

    Uses SERIALIZABLE isolation to prevent financial inconsistencies.
    """

    conn = psycopg2.connect(
        self.connection_string,
        cursor_factory=RealDictCursor
    )

    conn.set_isolation_level(isolation_level)

    try:
        yield conn

        conn.commit()

        logger.info("Transaction committed successfully")

    except Exception as e:
        conn.rollback()

        logger.error(f"Transaction rolled back due to error: {e}")

        raise

    finally:
        conn.close()

def execute_query(self, query: str, params=None):

    """Execute a read-only query and return results."""

    with self.transaction() as conn:

        cursor = conn.cursor()

        cursor.execute(query, params)
```

```
    return cursor.fetchall()

# Global database instance

db = DatabaseManager("postgresql://billing:password@localhost/billing_db")
```

Money and Currency Handling (`domain/value_objects.py`):

```
from decimal import Decimal, ROUND_HALF_UP

from dataclasses import dataclass

from typing import Union

import re

@dataclass(frozen=True)

class Money:

    """
    Immutable money value that maintains precision and currency information.

    Always stores amounts in the smallest currency unit (e.g., cents for USD).
    """

    amount_cents: int

    currency_code: str


    def __post_init__(self):

        if not re.match(r'^[A-Z]{3}$', self.currency_code):

            raise ValueError(f"Invalid currency code: {self.currency_code}")

        if not isinstance(self.amount_cents, int):

            raise ValueError("Amount must be stored as integer cents")


    @classmethod

    def from_decimal(cls, amount: Union[Decimal, float, str], currency: str) -> 'Money':

        """Create Money from decimal amount (e.g., 19.99 USD)."""

        decimal_amount = Decimal(str(amount))

        cents = int(decimal_amount.quantize(Decimal('0.01'), rounding=ROUND_HALF_UP) * 100)

        return cls(cents, currency.upper())
```

```
def to_decimal(self) -> Decimal:
    """Convert to decimal representation (e.g., 1999 cents -> 19.99)."""
    return Decimal(self.amount_cents) / 100


def __add__(self, other: 'Money') -> 'Money':
    if self.currency_code != other.currency_code:
        raise ValueError(f"Cannot add {self.currency_code} and {other.currency_code}")
    return Money(self.amount_cents + other.amount_cents, self.currency_code)


def __sub__(self, other: 'Money') -> 'Money':
    if self.currency_code != other.currency_code:
        raise ValueError(f"Cannot subtract {self.currency_code} and {other.currency_code}")
    return Money(self.amount_cents - other.amount_cents, self.currency_code)


def __mul__(self, factor: Union[int, Decimal]) -> 'Money':
    """Multiply money by a factor, rounding to nearest cent."""
    if isinstance(factor, int):
        return Money(self.amount_cents * factor, self.currency_code)

    decimal_result = Decimal(self.amount_cents) * Decimal(str(factor))
    rounded_cents = int(decimal_result.quantize(Decimal('1'), rounding=ROUND_HALF_UP))
    return Money(rounded_cents, self.currency_code)


def is_zero(self) -> bool:
    return self.amount_cents == 0
```

```
def is_positive(self) -> bool:
    return self.amount_cents > 0

def is_negative(self) -> bool:
    return self.amount_cents < 0

# Currency definitions for validation

SUPPORTED_CURRENCIES = {

    'USD': {'name': 'US Dollar', 'symbol': '$'},
    'EUR': {'name': 'Euro', 'symbol': '€'},
    'GBP': {'name': 'British Pound', 'symbol': '£'},
}

def validate_currency(currency_code: str) -> bool:
    """Validate that currency is supported by the system."""
    return currency_code.upper() in SUPPORTED_CURRENCIES
```

Event Logging for Audit Trails (`infrastructure/audit/event_logger.py`):

```
import json
```

PYTHON

```
import logging
```

```
from datetime import datetime, timezone
```

```
from typing import Any, Dict, Optional
```

```
from enum import Enum
```

```
class AuditEventType(Enum):
```

```
    SUBSCRIPTION_CREATED = "subscription_created"
```

```
    SUBSCRIPTION_UPDATED = "subscription_updated"
```

```
    PLAN_CHANGED = "plan_changed"
```

```
    PAYMENT_PROCESSED = "payment_processed"
```

```
    USAGE_RECORDED = "usage_recorded"
```

```
    INVOICE_GENERATED = "invoice_generated"
```

```
class AuditLogger:
```

```
    """
```

```
    Immutable audit trail for all billing operations.
```

```
    Required for compliance and debugging billing issues.
```

```
    """
```

```
def __init__(self, database_manager):
```

```
    self.db = database_manager
```

```
    self.logger = logging.getLogger(f"{__name__}.audit")
```

```
def log_event(
```

```
    self,
```

```
    event_type: AuditEventType,
```

```
    entity_id: str,
```

```
entity_type: str,  
  
changes: Optional[Dict[str, Any]] = None,  
  
actor_id: Optional[str] = None,  
  
metadata: Optional[Dict[str, Any]] = None  
):  
  
"""  
  
Log a billing operation event with complete context.  
  
All events are immutable and permanently stored.  
  
"""  
  
event_data = {  
  
    'event_id': self._generate_event_id(),  
  
    'timestamp': datetime.now(timezone.utc).isoformat(),  
  
    'event_type': event_type.value,  
  
    'entity_id': entity_id,  
  
    'entity_type': entity_type,  
  
    'actor_id': actor_id,  
  
    'changes': changes or {},  
  
    'metadata': metadata or {}  
}  
  
# Log to application logs for immediate visibility  
  
self.logger.info(f"Audit Event: {event_type.value}", extra=event_data)  
  
# Store in database for permanent audit trail  
  
self._store_audit_event(event_data)  
  
def _generate_event_id(self) -> str:
```

```
"""Generate unique event identifier."""

import uuid

return str(uuid.uuid4())


def _store_audit_event(self, event_data: Dict[str, Any]):

    """Store audit event in database audit table."""

    query = """

        INSERT INTO audit_events (

            event_id, timestamp, event_type, entity_id, entity_type,
            actor_id, changes, metadata

        ) VALUES (

            %(event_id)s, %(timestamp)s, %(event_type)s, %(entity_id)s,
            %(entity_type)s, %(actor_id)s, %(changes)s, %(metadata)s

        )

    """

    # Convert dictionaries to JSON strings for storage

    params = event_data.copy()

    params['changes'] = json.dumps(params['changes'])

    params['metadata'] = json.dumps(params['metadata'])

    try:

        with self.db.transaction() as conn:

            cursor = conn.cursor()

            cursor.execute(query, params)

    except Exception as e:

        # Audit logging failure should not break business operations
```

```
    self.logger.error(f"Failed to store audit event: {e}")

# Global audit logger instance
audit_logger = AuditLogger(db)
```

Core System Configuration

Application Settings (`config/settings.py`):

```
from decimal import Decimal

from typing import Dict, Any

import os

class BillingSettings:

    """Central configuration for billing system behavior."""

    # Database configuration

    DATABASE_URL = os.getenv('DATABASE_URL',
    'postgresql://billing:password@localhost/billing_db')

    # Payment gateway integration

    PAYMENT_GATEWAY_URL = os.getenv('PAYMENT_GATEWAY_URL', 'http://localhost:8001')

    PAYMENT_GATEWAY_API_KEY = os.getenv('PAYMENT_GATEWAY_API_KEY')

    # Billing cycle configuration

    DEFAULT_BILLING_ANCHOR = 1 # Day of month for monthly billing

    BILLING_GRACE_PERIOD_DAYS = 3

    MAX_PAYMENT_RETRIES = 3

    PAYMENT_RETRY_DELAYS = [1, 3, 7] # Days between retry attempts

    # Currency and precision settings

    DEFAULT_CURRENCY = 'USD'

    SUPPORTED_CURRENCIES = ['USD', 'EUR', 'GBP']

    DECIMAL_PRECISION = Decimal('0.01') # Cent precision

    # Usage billing configuration

    USAGE_AGGREGATION_WINDOW = 3600 # Seconds (1 hour)
```

```
MAX_USAGE_EVENTS_PER_REQUEST = 1000

USAGE_IDEMPOTENCY_WINDOW_HOURS = 24


# Trial and subscription defaults

DEFAULT_TRIAL_DAYS = 14

SUBSCRIPTION_REACTIVATION_GRACE_DAYS = 30


# System limits

MAX_PLAN_FEATURES = 50

MAX_PRICING_TIERS = 10


# Webhook configuration

WEBHOOK_TIMEOUT_SECONDS = 30

WEBHOOK_RETRY_ATTEMPTS = 5


@classmethod

def get_currency_config(cls, currency_code: str) -> Dict[str, Any]:
    """Get configuration for specific currency."""

    currency_configs = {

        'USD': {'symbol': '$', 'decimal_places': 2},

        'EUR': {'symbol': '€', 'decimal_places': 2},

        'GBP': {'symbol': '£', 'decimal_places': 2},
    }

    return currency_configs.get(currency_code, currency_configs['USD'])

settings = BillingSettings()
```

Development and Testing Setup

Docker Compose for Local Development (`docker-compose.yml`):

YAML

```
version: '3.8'

services:

  postgres:
    image: postgres:15
    environment:
      POSTGRES_DB: billing_db
      POSTGRES_USER: billing
      POSTGRES_PASSWORD: password
    ports:
      - "5432:5432"
    volumes:
      - billing_postgres_data:/var/lib/postgresql/data
      - ./scripts/init_db.sql:/docker-entrypoint-initdb.d/init_db.sql

  redis:
    image: redis:7-alpine
    ports:
      - "6379:6379"

  billing_api:
    build: .
    ports:
      - "8000:8000"
    environment:
      DATABASE_URL: postgresql://billing:password@postgres:5432/billing_db
      REDIS_URL: redis://redis:6379
    depends_on:
```

```
- postgres  
  
- redis  
  
volumes:  
  
- .:/app  
  
command: python -m uvicorn src.billing.api.main:app --host 0.0.0.0 --reload  
  
volumes:  
  
billing_postgres_data:
```

Key Development Patterns

Error Handling Strategy: The billing system implements a **fail-fast with detailed context** approach. Financial operations either complete successfully with full audit trails or fail immediately with sufficient information for debugging and customer communication.

Testing Philosophy: Every financial calculation includes property-based tests that verify mathematical properties (e.g., `upgrade_charge + downgrade_credit = 0` for immediate plan switches). Unit tests cover edge cases while integration tests validate complete billing workflows.

Monitoring and Alerting: The system logs all financial operations at INFO level with structured data, enabling real-time monitoring of billing accuracy and performance. Critical financial discrepancies trigger immediate alerts.

Development Checkpoint: After setting up this infrastructure, you should be able to start the application with `docker-compose up` and see a healthy PostgreSQL connection with proper decimal handling. Test the Money class with various currency calculations to verify precision handling works correctly.

Goals and Non-Goals

Milestone(s): Foundation for all milestones - establishes scope and requirements for the subscription billing system

Mental Model: Building a Financial Operating System

Think of the subscription billing system as a **financial operating system** for digital businesses. Just like how your computer's operating system manages resources, schedules tasks, and provides services to

applications, our billing system manages financial resources, schedules recurring payments, and provides billing services to business applications.

Consider Netflix's subscription model. When you sign up, the system must track your plan selection, manage your free trial period, automatically charge your payment method each month on your billing anniversary, handle plan upgrades when you switch from Basic to Premium, calculate prorated charges if you upgrade mid-month, track your viewing hours for potential usage-based features, generate invoices with proper tax calculations, retry failed payments with smart dunning logic, and gracefully handle cancellations while maintaining data integrity. This complexity multiplied across millions of subscribers requires a robust, reliable financial operating system.

The key insight is that subscription billing is not just about charging credit cards monthly. It's about managing complex financial relationships over time, handling state transitions reliably, maintaining audit trails for compliance, and ensuring every cent is accounted for with mathematical precision. Like an operating system, it must be reliable, performant, and extensible while hiding complexity from the applications that use it.

Functional Requirements

The subscription billing system must provide comprehensive lifecycle management capabilities that handle every aspect of the customer's financial journey from trial signup through cancellation and beyond.

Core Subscription Management

The system must support flexible subscription creation with multiple plan tiers and pricing models. Every customer must be able to select from available plans, with the system automatically provisioning their subscription with the correct billing cycle, feature entitlements, and payment schedule. The platform must handle multiple billing intervals including monthly, quarterly, and annual cycles, with proper calendar handling for edge cases like month-end dates.

Capability	Description	Business Rules
Plan Selection	Customer chooses from available subscription tiers	Must validate plan availability and currency support
Trial Management	Free trial periods with automatic conversion	Trial length configurable per plan, requires payment method on file
Billing Cycles	Multiple billing intervals (monthly, quarterly, annual)	Billing anchor date preserved across cycles, handle calendar edge cases
Feature Provisioning	Grant access based on subscription tier	Real-time feature flag updates, immediate access changes
Multi-Currency Support	Global customer base with local currencies	Support major currencies, handle exchange rate fluctuations

Plan and Pricing Flexibility

The system must accommodate diverse pricing strategies that businesses use to optimize revenue. This includes flat-rate subscriptions, per-seat pricing for team plans, tiered usage allowances, and volume-based discounts. Plan management must support versioning to protect existing customers while allowing businesses to evolve their pricing strategies.

Pricing Model	Implementation Requirements	Edge Cases
Flat-Rate	Fixed monthly/annual fee	Handle currency precision, tax calculations
Per-Seat	Base price plus per-user charges	Seat additions/removals mid-cycle, minimum seat requirements
Tiered Usage	Included allowance plus overage rates	Usage tracking accuracy, billing period boundaries
Volume Discounts	Price breaks at usage thresholds	Retroactive discounts vs progressive pricing
Custom Contracts	Enterprise-specific pricing terms	Contract versioning, approval workflows

Subscription Lifecycle Transitions

The system must handle every possible subscription state change with proper validation, audit logging, and downstream effects. State transitions must be atomic to prevent inconsistent states, and all changes must trigger appropriate notifications and integrations.

State Transition	Trigger Conditions	Required Actions
Trial → Active	Trial period expires, valid payment method	Process first payment, activate features, send welcome email
Active → Past Due	Payment failure	Disable features based on grace period policy, start dunning sequence
Past Due → Active	Successful payment retry	Restore full access, clear past due status, update billing cycle
Active → Cancelled	Customer or admin cancellation	Calculate final charges, schedule termination, retain data per policy
Cancelled → Active	Reactivation within grace period	Restore subscription state, recalculate billing, reactivate features

Proration and Plan Changes

Mid-cycle plan changes require sophisticated proration calculations that account for time remaining in the billing period, feature usage to date, and credit management. The system must support both immediate and scheduled plan changes, with proper financial reconciliation in all cases.

Critical Design Principle: Proration calculations must be deterministic and auditable. The same inputs must always produce identical outputs, and every calculation step must be logged for customer service and compliance purposes.

Change Type	Calculation Method	Credit Handling
Upgrade	Prorated charge for price difference	Immediate charge to payment method
Downgrade	Credit for unused higher-tier value	Apply credit to customer balance for future invoices
Quantity Increase	Prorated charge for additional seats/units	Immediate charge based on remaining billing period
Quantity Decrease	Credit for removed seats/units	Credit application with configurable timing

Usage-Based Billing Integration

For products with consumption-based pricing, the system must accurately track usage events, aggregate them across billing periods, and convert usage into billable charges. Usage tracking must be idempotent to handle duplicate events, and aggregation must be mathematically precise to prevent revenue leakage or customer disputes.

The usage engine must support multiple aggregation methods including sum (total API calls), count (number of transactions), maximum (peak concurrent users), and last-value (storage at end of period). Each aggregation method serves different billing models and requires different handling of late-arriving data.

Usage Pattern	Aggregation Method	Billing Calculation
API Calls	Sum across billing period	Total calls × per-call rate + overage tiers
Storage	Last value in period	GB-months × storage rate
Concurrent Users	Maximum daily peak	Peak users × per-user rate
Transactions	Count with deduplication	Transaction count × per-transaction fee
Bandwidth	Sum with time weighting	Total GB × bandwidth rate + burst charges

Non-Functional Requirements

The billing system operates in a high-stakes environment where errors directly impact revenue and customer trust. Non-functional requirements are not optional nice-to-haves but essential characteristics that determine system success.

Financial Precision and Data Integrity

All monetary calculations must maintain cent-level precision throughout the system lifecycle. The system must use the `Money` value object with fixed-point arithmetic to eliminate floating-point precision errors that could accumulate into significant revenue discrepancies over millions of transactions.

Precision Requirement	Implementation Standard	Validation Method
Currency Calculations	Store all amounts in smallest currency unit (cents)	Automated reconciliation against payment processor
Proration Math	Use <code>DECIMAL_PRECISION</code> constant for rounding	Unit tests with known edge cases, property-based testing
Tax Calculations	Round per jurisdiction rules	Integration tests with tax service providers
Exchange Rates	Daily rate updates with historical tracking	Compare against multiple rate sources
Audit Trail	Every calculation step logged	End-to-end audit reports match transaction details

Performance and Scalability Targets

The system must handle enterprise-scale workloads with predictable performance characteristics. Billing operations have natural batch processing windows (monthly cycles) but also require real-time responsiveness for customer-facing operations like plan changes and usage queries.

Operation Type	Performance Target	Scalability Requirement
Plan Changes	< 2 seconds end-to-end	Handle 1000 concurrent plan changes
Usage Event Ingestion	< 100ms per event, 10K events/second	Linear scaling with processing nodes
Monthly Billing Run	Process 100K subscriptions in 4 hours	Parallel processing with progress tracking
Invoice Generation	< 5 seconds per invoice	Support 1M+ invoices per billing cycle
Payment Processing	< 10 seconds including gateway roundtrip	Graceful degradation during gateway issues

Reliability and Fault Tolerance

Billing system failures can result in lost revenue, duplicate charges, or compliance violations. The system must be designed for high availability with comprehensive error recovery and graceful degradation capabilities.

Reliability Principle: Every billing operation must be either fully completed or safely rolled back. Partial states that could result in financial discrepancies are unacceptable.

Failure Scenario	Detection Method	Recovery Strategy
Database Connection Loss	Health checks every 30 seconds	Automatic failover to read replica, queue writes
Payment Gateway Timeout	Request timeout after 30 seconds	Retry with exponential backoff, manual reconciliation queue
Proration Calculation Error	Input validation and range checks	Fallback to manual approval workflow
Usage Data Corruption	Checksum validation on aggregation	Rebuild from raw events, alert operations team
Billing Cycle Processing Failure	Progress tracking and heartbeat monitoring	Resume from last checkpoint, skip completed records

Security and Compliance Standards

The billing system handles sensitive financial data and must meet stringent security requirements including PCI DSS Level 1 compliance for payment card data handling. All financial operations must maintain comprehensive audit trails for regulatory compliance and customer dispute resolution.

Security Domain	Requirement	Implementation
Data Encryption	All PII and financial data encrypted at rest and in transit	AES-256 encryption, TLS 1.3 for transport
Access Control	Role-based permissions with principle of least privilege	Integration with identity provider, session management
Audit Logging	Every billing operation logged with actor identification	Immutable audit log, tamper detection
PCI Compliance	Never store payment card data in billing database	Token-based payment references, annual compliance audit
Data Retention	Customer data retention per privacy regulations	Automated data lifecycle management, right to deletion

Explicit Non-Goals

Clearly defining what the subscription billing system will **not** handle is crucial for maintaining focus and preventing scope creep. These exclusions represent deliberate architectural decisions that keep the system focused on its core competencies.

Payment Processing Infrastructure

The billing system will **not** implement direct payment processing capabilities such as credit card authorization, fraud detection, or PCI-compliant card data storage. These capabilities are provided by the prerequisite payment gateway system, which handles all sensitive payment operations and compliance requirements.

Excluded Capability	Rationale	Alternative Approach
Credit Card Processing	Requires PCI Level 1 compliance infrastructure	Integrate with existing payment gateway
Fraud Detection	Specialized domain requiring ML models and risk databases	Use payment processor's fraud services
Payment Method Storage	Complex PCI compliance and security requirements	Store tokenized references only
Chargeback Management	Requires specialized dispute handling processes	Payment gateway handles disputes
Bank Account Verification	Complex regulatory requirements vary by jurisdiction	Use payment processor's ACH services

Advanced Revenue Recognition

The system will not implement complex revenue recognition rules required for GAAP or IFRS financial reporting. While the system tracks subscription revenue and provides basic reporting, sophisticated revenue recognition with contract modifications, performance obligations, and accounting standard compliance is outside scope.

Boundary Decision: The billing system focuses on operational billing accuracy rather than accounting compliance. Financial reporting requirements should be handled by dedicated accounting systems that consume billing data.

Complex Tax Calculation

The system will not implement comprehensive tax calculation engines for sales tax, VAT, or other jurisdiction-specific tax requirements. Basic tax rate application is supported, but complex tax scenarios such as tax nexus determination, exemption certificate management, and regulatory filing are handled by specialized tax services.

Tax Complexity	System Support	External Integration
Simple Tax Rates	Apply configured rates per jurisdiction	Built-in capability
Tax Nexus Rules	Not supported	Integrate with tax calculation service
Exemption Certificates	Not supported	Use tax service certificate management
Regulatory Filing	Not supported	Export data to tax compliance system
Multi-jurisdictional Compliance	Not supported	Tax service handles compliance

Enterprise Contract Management

The system will not support complex enterprise contract features such as custom approval workflows, contract negotiation tracking, or sophisticated discount approval chains. While the system supports flexible pricing and custom plans, it assumes these configurations are managed through administrative interfaces rather than complex contract lifecycle management.

Multi-Tenant Architecture

The initial system design assumes a single-tenant deployment model. While the data model and service architecture are designed to be tenant-aware for future extension, the system will not initially support multiple isolated customer organizations within a single deployment instance.

Real-Time Analytics and Business Intelligence

The system will not include sophisticated analytics dashboards, revenue forecasting, or business intelligence capabilities. While basic subscription metrics and billing reports are provided, advanced analytics should be handled by dedicated business intelligence systems that consume billing data through APIs.

Analytics Type	System Capability	External Tool Required
Basic Subscription Metrics	Count active subscriptions, MRR calculation	Built-in reporting
Customer Lifecycle Analysis	Not supported	Business intelligence platform
Revenue Forecasting	Not supported	Analytics platform
Cohort Analysis	Not supported	Data warehouse and BI tools
Churn Prediction	Not supported	Machine learning platform

Implementation Guidance

The subscription billing system requires careful technology selection to handle the demanding requirements of financial precision, high availability, and regulatory compliance.

Technology Recommendations

Component	Simple Option	Advanced Option
Web Framework	Flask with SQLAlchemy	FastAPI with async database drivers
Database	PostgreSQL with financial precision types	PostgreSQL with read replicas and connection pooling
Message Queue	Redis with pub/sub	Apache Kafka for high-throughput event streaming
Caching	Redis with TTL-based invalidation	Redis Cluster with consistent hashing
API Documentation	OpenAPI/Swagger auto-generation	AsyncAPI for event-driven APIs
Testing Framework	pytest with factory patterns	pytest with property-based testing (Hypothesis)
Monitoring	Prometheus with custom business metrics	Datadog with financial reconciliation dashboards

Recommended Project Structure

```
subscription-billing/
├── src/
│   ├── billing/
│   │   ├── __init__.py
│   │   └── domain/           ← Core business entities
│   │       ├── entities/
│   │       │   ├── __init__.py
│   │       │   ├── money.py      ← Money value object
│   │       │   ├── customer.py
│   │       │   ├── plan.py
│   │       │   └── subscription.py
│   │       ├── services/        ← Business logic services
│   │       │   ├── plan_manager.py
│   │       │   ├── subscription_engine.py
│   │       │   ├── proration_calculator.py
│   │       │   └── usage_tracker.py
│   │       └── repositories/    ← Data access abstractions
│   ├── infrastructure/        ← External integrations
│   │   ├── database/
│   │   │   ├── __init__.py
│   │   │   ├── models.py       ← SQLAlchemy models
│   │   │   └── migrations/
│   │   ├── payment_gateway/
│   │   └── notifications/
│   ├── api/                  ← HTTP API layer
│   │   ├── __init__.py
│   │   ├── routes/
│   │   └── schemas/
│   └── tasks/                ← Background job processing
│       ├── billing_cycle.py
│       └── usage_aggregation.py
└── tests/
    ├── unit/
    ├── integration/
    └── fixtures/
└── migrations/             ← Database schema evolution
└── config/
    ├── development.py
    ├── production.py
    └── test.py
└── requirements/
    ├── base.txt
    ├── development.txt
    └── production.txt
└── docker/
    ├── Dockerfile
    └── docker-compose.yml
```

Core Infrastructure Components

Money Value Object Implementation:

```
# src/billing/domain/entities/money.py

from decimal import Decimal, ROUND_HALF_UP

from typing import Dict

import attr


# Financial precision constant - always round to nearest cent

DECIMAL_PRECISION = Decimal('0.01')

DEFAULT_CURRENCY = 'USD'

# Supported currencies with metadata

SUPPORTED_CURRENCIES = {

    'USD': {'symbol': '$', 'decimal_places': 2, 'name': 'US Dollar'},

    'EUR': {'symbol': '€', 'decimal_places': 2, 'name': 'Euro'},

    'GBP': {'symbol': '£', 'decimal_places': 2, 'name': 'British Pound'},

    'CAD': {'symbol': 'C$', 'decimal_places': 2, 'name': 'Canadian Dollar'},

    'JPY': {'symbol': '¥', 'decimal_places': 0, 'name': 'Japanese Yen'},

}

@attr.s(frozen=True, auto_attribs=True)

class Money:

    """Immutable value object for financial amounts.

    Stores amounts as integer cents to avoid floating-point precision issues.

    All arithmetic operations return new Money instances.

    """

    amount_cents: int

    currency_code: str = DEFAULT_CURRENCY
```

PYTHON

```
def __attrs_post_init__(self):
    if self.currency_code not in SUPPORTED_CURRENCIES:
        raise ValueError(f"Unsupported currency: {self.currency_code}")

    if not isinstance(self.amount_cents, int):
        raise TypeError("Amount must be stored as integer cents")

@classmethod
def from_decimal(cls, amount: Decimal, currency: str = DEFAULT_CURRENCY) -> 'Money':
    """Create Money from decimal amount, converting to cents."""
    # TODO: Round to appropriate precision for currency
    # TODO: Convert to integer cents (multiply by 100 for most currencies)
    # TODO: Handle zero-decimal currencies like JPY
    # TODO: Validate amount is not negative for billing purposes
    pass

def to_decimal(self) -> Decimal:
    """Convert to decimal representation for display and calculations."""
    # TODO: Divide cents by appropriate factor based on currency
    # TODO: Return Decimal with proper precision
    # TODO: Handle zero-decimal currencies
    pass

def add(self, other: 'Money') -> 'Money':
    """Add two Money amounts, must be same currency."""
    # TODO: Validate currencies match
    # TODO: Add amounts in cents
    # TODO: Return new Money instance
```

```
pass

def subtract(self, other: 'Money') -> 'Money':
    """Subtract other from this amount."""
    # TODO: Validate currencies match
    # TODO: Subtract in cents
    # TODO: Allow negative results for credits/refunds
    pass

def multiply(self, factor: Decimal) -> 'Money':
    """Multiply amount by decimal factor, used for proration."""
    # TODO: Multiply cents by factor
    # TODO: Round to nearest cent using ROUND_HALF_UP
    # TODO: Return new Money instance
    pass
```

Database Transaction Management:

```
# src/billing/infrastructure/database/__init__.py

from contextlib import contextmanager

from sqlalchemy import create_engine

from sqlalchemy.orm import sessionmaker

import logging

logger = logging.getLogger(__name__)

# Serializable isolation for financial operations

ISOLATION_LEVEL_SERIALIZABLE = 'SERIALIZABLE'

class DatabaseManager:

    """Manages database connections and transactions for billing operations."""

    def __init__(self, connection_string: str):

        self.connection_string = connection_string

        self.engine = None

        self.SessionFactory = None

        # TODO: Initialize SQLAlchemy engine with connection pooling

        # TODO: Configure session factory with appropriate defaults

        # TODO: Set up connection health checking

    @contextmanager

    def transaction(self, isolation_level: str = None):

        """Provide transactional context for billing operations.

        Args:

            isolation_level: Database isolation level, defaults to READ_COMMITTED
```

```
    Use ISOLATION_LEVEL_SERIALIZABLE for financial operations

    """
    session = self.SessionFactory()

    if isolation_level:

        session.connection(execution_options={"isolation_level": isolation_level})



    try:

        # TODO: Begin transaction

        # TODO: Yield session to calling code

        # TODO: Commit transaction on success

        # TODO: Log successful transaction for audit

        pass

    except Exception as e:

        # TODO: Rollback transaction on any error

        # TODO: Log rollback details for debugging

        # TODO: Re-raise exception for handling by calling code

        pass

    finally:

        # TODO: Always close session to return to pool

        pass
```

Audit Logging Infrastructure:

```
# src/billing/domain/services/audit_logger.py
```

PYTHON

```
from enum import Enum

from datetime import datetime

from typing import Dict, Any, Optional

import json


class AuditEventType(Enum):

    """Categories for billing system audit events."""

    SUBSCRIPTION_CREATED = "subscription_created"

    SUBSCRIPTION_CANCELLED = "subscription_cancelled"

    PLAN_CHANGED = "plan_changed"

    PAYMENT_PROCESSED = "payment_processed"

    PRORATION_CALCULATED = "proration_calculated"

    USAGE_RECORDED = "usage_recorded"

    INVOICE_GENERATED = "invoice_generated"


class AuditLogger:

    """Records immutable audit trail for all billing operations."""

    def __init__(self, database_manager: DatabaseManager):

        self.db = database_manager


    def log_event(

        self,

        event_type: AuditEventType,

        entity_id: str,

        entity_type: str,
```

```
    changes: Dict[str, Any],  
  
    actor_id: str,  
  
    metadata: Optional[Dict[str, Any]] = None  
 ) -> None:  
  
    """Record audit event for billing operation.  
  
  
Args:  
  
    event_type: Type of operation being audited  
  
    entity_id: ID of the entity being modified  
  
    entity_type: Type of entity (subscription, invoice, etc.)  
  
    changes: Before/after values for the operation  
  
    actor_id: ID of user or system performing the operation  
  
    metadata: Additional context for the operation  
  
    """  
  
    # TODO: Create audit record with timestamp and unique ID  
  
    # TODO: Serialize changes and metadata to JSON  
  
    # TODO: Insert into audit_events table using serializable transaction  
  
    # TODO: Handle any insertion errors without affecting main operation  
  
    # TODO: Log audit record creation for debugging  
  
    pass
```

Configuration Management

```
# src/billing/config.py                                         PYTHON

from decimal import Decimal

import os

class BillingSettings:

    """Configuration settings for the billing system."""

    # Financial precision settings

    DECIMAL_PRECISION = Decimal('0.01')

    DEFAULT_CURRENCY = 'USD'

    # Billing cycle settings

    BILLING_GRACE_PERIOD_DAYS = 7

    DUNNING_RETRY_ATTEMPTS = 3

    PRORATION_ROUNDING_METHOD = 'ROUND_HALF_UP'

    # Performance settings

    USAGE_BATCH_SIZE = 1000

    BILLING_PROCESSING_CHUNK_SIZE = 500

    # Integration settings

    PAYMENT_GATEWAY_TIMEOUT = 30

    WEBHOOK_RETRY_ATTEMPTS = 5

    @classmethod

        def from_environment(cls) -> 'BillingSettings':
```

```
"""Load settings from environment variables."""

settings = cls()

# TODO: Override defaults with environment variables

# TODO: Validate required settings are present

# TODO: Type conversion and validation for numeric settings

# TODO: Return configured settings instance

return settings
```

Milestone Validation Checkpoints

Checkpoint 1 - Core Infrastructure: Run `python -m pytest tests/unit/test_money.py -v` to verify the `Money` value object handles all currency operations correctly. Expected output should show all precision tests passing, including edge cases like currency conversion and proration calculations.

Checkpoint 2 - Database Setup: Execute `python -m billing.infrastructure.database.setup_test_data` to create sample plans and subscriptions. Verify you can query the database and see properly formatted monetary values stored as integers.

Checkpoint 3 - Requirements Validation: Create a simple test subscription through the API and verify all functional requirements are testable:

- Plan selection creates subscription with correct billing cycle
- Trial period is properly configured and tracked
- Payment method is tokenized and stored securely
- Audit events are created for all operations

Common Configuration Pitfalls

⚠ Pitfall: Floating-Point Currency Storage Never store monetary amounts as floating-point numbers in the database. Use integer cents or decimal types with fixed precision. Floating-point arithmetic can introduce rounding errors that compound over thousands of transactions.

⚠ Pitfall: Insufficient Audit Logging Billing systems require comprehensive audit trails for compliance and dispute resolution. Log every state change, calculation, and external API call with sufficient detail to reconstruct the operation later.

⚠ Pitfall: Inadequate Error Handling Billing operations must be atomic - either fully completed or fully rolled back. Partial states where money has been charged but subscription hasn't been activated can result in customer disputes and revenue recognition issues.

⚠ Pitfall: Time Zone Confusion Always store timestamps in UTC and convert to customer's local timezone for display only. Billing cycles based on local times can create confusion when customers travel or daylight saving time changes occur.

High-Level Architecture

Milestone(s): Foundation for all milestones - provides the overall system structure that supports plan management, subscription lifecycle, proration, and usage-based billing

Component Overview

Mental Model: The Financial Operations Center

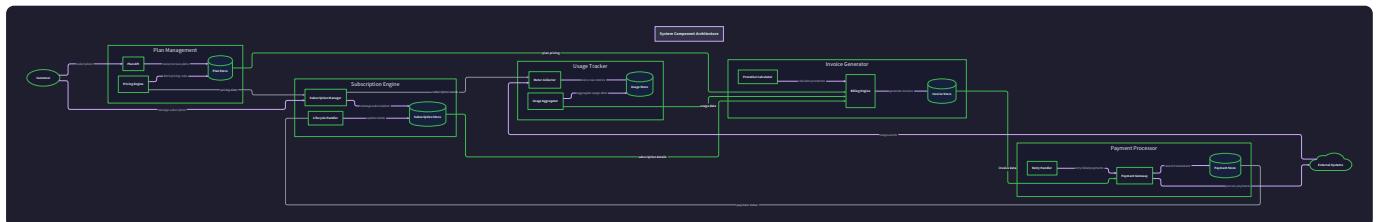
Think of the subscription billing system like a modern financial operations center at a major bank. Just as a bank has specialized departments—customer accounts, loan processing, transaction monitoring, payment clearing, and audit—our billing system divides complex financial operations into focused, specialized components that work together seamlessly.

The **Plan Management** component acts like the bank's product catalog department, defining what financial products (subscription plans) are available, their terms, and who can access them. The **Subscription Engine** functions like the account management department, tracking each customer's relationship with the bank, their account status, and lifecycle events. The **Usage Tracker** operates like the transaction monitoring system, recording every billable event in real-time. The **Invoice Generator** works like the billing department, calculating what each customer owes and producing formal billing statements. Finally, the **Payment Processor** acts like the payment clearing house, handling the actual movement of money.

Each department has clear responsibilities and communicates through well-defined channels, but they're all orchestrated to create a seamless customer experience. When a customer upgrades their plan, multiple departments coordinate: product catalog validates the new plan, account management updates the relationship, billing calculates proration, and payment processing handles any immediate charges.

The subscription billing system follows this same pattern of specialized components with clear boundaries but coordinated workflows.

Core System Components



The subscription billing system consists of six major components, each with distinct responsibilities and clear interfaces:

Plan Management Component

The Plan Management component serves as the product catalog and pricing engine for the entire system. It maintains the canonical definition of all subscription plans, their pricing tiers, feature entitlements, and billing parameters. This component handles plan versioning to ensure existing customers retain their original terms while new customers receive updated plans. It validates plan configurations for consistency and provides the pricing logic used throughout the system.

Responsibility	Description	Data Owned
Plan Definition	Maintains subscription plans with pricing, features, and terms	Plan configurations, pricing tiers, feature matrices
Pricing Logic	Calculates base charges for different plan types and billing intervals	Pricing formulas, currency conversions, discount rules
Feature Entitlements	Maps plan tiers to accessible features and usage limits	Feature flags, quota definitions, access control lists
Plan Versioning	Manages plan changes while protecting existing customer terms	Plan versions, deprecation schedules, grandfathering rules

Subscription Engine

The Subscription Engine manages the complete customer subscription lifecycle from creation through cancellation. It maintains the subscription state machine, processes lifecycle events like renewals and cancellations, and coordinates with other components during plan changes. This component serves as the system of record for all subscription relationships and their current status.

Responsibility	Description	Data Owned
Lifecycle Management	Handles subscription creation, renewal, cancellation, and reactivation	Subscription records, state transitions, lifecycle events
State Machine	Enforces valid subscription state transitions and business rules	Current states, transition logs, business rule validations
Billing Orchestration	Coordinates billing cycles and triggers invoice generation	Billing schedules, anniversary dates, renewal triggers
Plan Changes	Processes upgrades and downgrades with effective date management	Change requests, effective dates, approval workflows

Usage Tracker

The Usage Tracker implements event-based metering for usage-based billing scenarios. It ingests usage events with idempotent processing, aggregates consumption data over billing periods, and calculates usage-based charges according to plan terms. This component ensures accurate usage measurement and prevents double-billing through robust deduplication.

Responsibility	Description	Data Owned
Event Ingestion	Accepts and validates usage events with idempotency protection	Usage events, deduplication keys, ingestion timestamps
Usage Aggregation	Sums, counts, or calculates max usage over billing periods	Aggregated usage totals, billing period boundaries, calculation results
Overage Calculation	Determines charges when usage exceeds plan allowances	Usage limits, overage rates, tiered pricing calculations
Real-time Quotas	Tracks current usage against plan limits for quota enforcement	Current usage counters, quota thresholds, limit enforcement

Proration Calculator

The Proration Calculator handles all partial charge and credit calculations when subscriptions change mid-cycle. It implements precise financial calculations for time-based and quantity-based proration, manages customer credit balances, and ensures accurate billing during plan transitions. This component is critical for maintaining customer trust through transparent and accurate billing adjustments.

Responsibility	Description	Data Owned
Proration Logic	Calculates partial charges and credits for mid-cycle changes	Proration formulas, time calculations, rounding rules
Credit Management	Tracks and applies customer credit balances	Credit balances, credit applications, expiration policies
Change Processing	Handles immediate and scheduled plan changes with correct timing	Change schedules, effective dates, processing queues
Financial Precision	Ensures accurate monetary calculations with proper rounding	Precision settings, rounding algorithms, audit trails

Invoice Generator

The Invoice Generator creates formal billing statements by combining subscription charges, usage fees, proration adjustments, and credit applications. It produces invoices in multiple formats, handles tax calculations, and maintains a complete audit trail of all billing activity. This component ensures compliance with accounting standards and provides customers with clear, detailed billing statements.

Responsibility	Description	Data Owned
Invoice Creation	Generates invoices from subscription charges, usage, and adjustments	Invoice templates, line items, tax calculations
Line Item Assembly	Combines base charges, usage fees, proration, and credits into invoice lines	Charge breakdowns, itemized billing, descriptive labels
Tax Calculation	Applies appropriate tax rates based on customer location and plan type	Tax rates, jurisdiction rules, exemption handling
Audit Trail	Maintains complete financial audit trail for compliance	Invoice history, change logs, financial reconciliation data

Payment Processor

The Payment Processor integrates with the prerequisite payment gateway system to handle actual money movement. It processes recurring charges, manages payment methods, handles payment failures with retry logic, and processes webhooks from payment providers. This component abstracts payment complexity and provides consistent payment handling across the billing system.

Responsibility	Description	Data Owned
Payment Execution	Processes charges through payment gateways with retry logic	Payment attempts, transaction records, retry schedules
Method Management	Handles customer payment methods and updates	Payment methods, expiration tracking, update notifications
Webhook Processing	Processes payment gateway webhooks with idempotency	Webhook events, processing status, duplicate prevention
Failure Handling	Manages payment failures with dunning and grace periods	Failure reasons, retry attempts, dunning workflows

Component Interaction Patterns

The components interact through well-defined interfaces using event-driven and synchronous communication patterns. Each component publishes domain events when significant state changes occur, allowing other components to react appropriately without tight coupling.

Event-Driven Communication

Components publish events for major state changes that other components need to know about. This ensures loose coupling while maintaining system consistency. Events are processed asynchronously with at-least-once delivery guarantees.

Event Type	Publisher	Subscribers	Purpose
SubscriptionCreated	Subscription Engine	Usage Tracker, Invoice Generator	Initialize tracking and billing for new subscription
SubscriptionCancelled	Subscription Engine	Usage Tracker, Payment Processor	Stop usage tracking and cancel recurring payments
PlanChanged	Subscription Engine	Proration Calculator, Usage Tracker	Trigger proration and update usage limits
UsageReported	Usage Tracker	Invoice Generator	Include usage charges in next billing cycle
PaymentFailed	Payment Processor	Subscription Engine	Trigger dunning process and state changes
InvoiceGenerated	Invoice Generator	Payment Processor	Trigger payment processing for new invoices

Synchronous API Communication

For operations requiring immediate consistency or validation, components communicate through synchronous APIs. These interactions are typically read-only queries or validation requests that need immediate responses.

API Call	Caller	Provider	Purpose
ValidatePlanChange	Subscription Engine	Plan Management	Validate plan upgrade/downgrade before processing
CalculateProration	Subscription Engine	Proration Calculator	Get proration amounts for plan changes
GetUsageTotal	Invoice Generator	Usage Tracker	Retrieve usage charges for invoice generation
GetCreditBalance	Invoice Generator	Proration Calculator	Include customer credits in invoice
ProcessPayment	Invoice Generator	Payment Processor	Execute payment for new invoice

Integration Patterns

Mental Model: The Banking Network

Think of how banks integrate with each other and external financial systems. Your local bank doesn't operate in isolation—it connects to credit card networks, clearing houses, federal reserve systems, and other banks

through standardized protocols and interfaces. Each connection has specific purposes, security requirements, and failure handling procedures.

Similarly, our subscription billing system must integrate with external payment providers, tax calculation services, customer management systems, and business intelligence tools. Each integration requires careful design of interfaces, error handling, and data consistency guarantees.

Just as banks use different communication methods for different purposes—real-time wire transfers use secure direct connections while batch processing uses file transfers—our billing system uses different integration patterns based on the requirements of each external system.

Payment Gateway Integration

The subscription billing system integrates with the prerequisite payment gateway system through both API calls and webhook processing. This integration handles the actual movement of money while the billing system focuses on subscription logic and financial calculations.

Decision: Payment Gateway Integration Architecture

- **Context:** The billing system needs to process payments but should not handle sensitive payment data directly due to PCI compliance requirements
- **Options Considered:**
 1. Direct payment processing with PCI compliance
 2. Integration with existing payment gateway
 3. Third-party billing service integration
- **Decision:** Integrate with the prerequisite payment gateway system
- **Rationale:** Leverages existing payment infrastructure, maintains PCI compliance, and focuses billing system on subscription-specific logic rather than payment processing
- **Consequences:** Enables faster development and better security but creates dependency on payment gateway availability and capabilities

API Integration Pattern

The Payment Processor component communicates with the payment gateway through RESTful APIs for payment method management, charge processing, and customer management. All API calls include idempotency keys to prevent duplicate processing during retries.

Operation	Method	Purpose	Idempotency
Create Customer	POST /customers	Create gateway customer record	Customer ID as idempotency key
Add Payment Method	POST /customers/{id}/payment-methods	Attach payment method to customer	Method fingerprint as key
Process Charge	POST /charges	Execute one-time or recurring payment	Internal transaction ID as key
Refund Payment	POST /refunds	Process refunds for plan downgrades	Refund request ID as key

Webhook Integration Pattern

The payment gateway sends webhooks for asynchronous events like payment success, failure, and disputes. The billing system processes these webhooks with careful attention to idempotency, ordering, and failure handling.

Payment Gateway → Webhook Endpoint → Event Queue → Payment Processor → Subscription Engine

The webhook processing follows a reliable pattern:

1. **Immediate Acknowledgment:** Webhook endpoint immediately returns 200 OK after basic validation
2. **Event Queuing:** Valid webhooks are queued for asynchronous processing
3. **Idempotent Processing:** Each webhook event is processed exactly once using event IDs
4. **State Reconciliation:** Processing updates subscription states based on payment outcomes
5. **Retry Logic:** Failed webhook processing retries with exponential backoff

Customer Management Integration

The billing system integrates with existing customer management systems to maintain consistent customer data and avoid duplicating customer information. This integration typically follows an event-driven pattern where customer changes trigger updates in the billing system.

Customer Data Synchronization

Integration Type	Direction	Data Synchronized	Trigger
Customer Creation	CRM → Billing	Customer ID, contact info, billing address	New customer signup
Profile Updates	CRM → Billing	Address changes, contact updates	Customer profile modification
Subscription Events	Billing → CRM	Subscription status, plan changes	Billing lifecycle events
Usage Analytics	Billing → CRM	Usage patterns, billing history	Reporting and analytics needs

Tax Calculation Integration

For businesses operating in multiple jurisdictions, the billing system integrates with tax calculation services to ensure compliance with local tax regulations. This integration is particularly important for usage-based billing where tax calculations may be complex.

Tax Service Integration Pattern

The Invoice Generator component calls tax calculation services during invoice generation to determine appropriate tax amounts based on customer location, product type, and billing amounts.

Tax Scenario	Integration Point	Data Sent	Data Received
Subscription Charges	Invoice generation	Customer address, plan type, charge amount	Tax rate, tax amount, tax jurisdiction
Usage Charges	Usage billing	Usage type, consumption amount, customer location	Applicable tax rates and amounts
Proration Adjustments	Plan changes	Original charge, prorated amount, effective dates	Adjusted tax calculations
Refunds	Cancellations	Original tax amount, refund reason, jurisdiction	Tax refund amount and handling

Business Intelligence Integration

The billing system provides data to business intelligence and analytics platforms through both real-time events and batch data exports. This enables revenue reporting, customer analytics, and business performance monitoring.

Analytics Data Flow

Data Type	Export Method	Frequency	Contents
Revenue Events	Real-time streaming	Immediate	Invoice generation, payment success, refunds
Usage Metrics	Batch export	Daily	Aggregated usage by customer, plan, and feature
Subscription Analytics	Batch export	Daily	Churn analysis, upgrade/downgrade patterns, lifecycle metrics
Financial Reconciliation	Batch export	Monthly	Complete financial audit trail for accounting

External Service Error Handling

All external integrations implement robust error handling and fallback strategies to ensure billing operations continue even when external services are unavailable.

Critical Design Principle: External service failures should degrade functionality gracefully rather than preventing core billing operations

Error Handling Strategy

Service	Error Type	Immediate Action	Fallback Strategy	Recovery Action
Payment Gateway	Network timeout	Retry with exponential backoff	Queue payment for later processing	Process when service recovers
Tax Service	Service unavailable	Use cached tax rates	Apply default tax rate with adjustment flag	Recalculate when service returns
Customer Service	Data sync failure	Log inconsistency	Continue with local customer data	Sync when connectivity restored
Analytics	Export failure	Queue events locally	Continue billing operations	Replay events when service available

Recommended Codebase Structure

Mental Model: The Corporate Organizational Chart

Think of a well-organized corporation with clear departments, reporting structures, and communication channels. The accounting department doesn't directly manage customer service, but they work together through defined processes. Similarly, our codebase structure reflects the business domain organization while maintaining clean separation of concerns.

Just as a company has shared services (IT, legal, facilities) that support all departments, our codebase has common infrastructure (database, logging, configuration) that supports all business components. The directory structure should make it immediately clear which code handles which business responsibility, just like an organizational chart shows who handles which business function.

Directory Structure Organization

The codebase follows a domain-driven design approach with clear separation between business logic, infrastructure concerns, and external interfaces. This structure supports both the current requirements and future scalability needs.

```
subscription-billing-system/
├── cmd/                                # Application entry points
│   ├── billing-server/                  # Main billing API server
│   │   └── main.py
│   ├── usage-ingestion/                # Usage event ingestion service
│   │   └── main.py
│   └── billing-scheduler/              # Batch billing operations
│       └── main.py
├── internal/                            # Internal business logic (not importable)
│   ├── domain/                          # Core business entities and rules
│   │   ├── customer/                   # Customer entity and value objects
│   │   │   ├── __init__.py
│   │   │   ├── models.py
│   │   │   └── repository.py          # Customer data access interface
│   │   ├── plan/                      # Plan entities and pricing logic
│   │   │   ├── __init__.py
│   │   │   ├── models.py
│   │   │   ├── repository.py          # Plan data access interface
│   │   │   └── service.py            # Plan management business logic
│   │   ├── subscription/             # Subscription state machine and lifecycle
│   │   │   ├── __init__.py
│   │   │   ├── models.py
│   │   │   ├── repository.py          # Subscription persistence interface
│   │   │   └── service.py            # Subscription lifecycle management
│   │   ├── usage/                    # Usage events and aggregation
│   │   │   ├── __init__.py
│   │   │   ├── models.py
│   │   │   ├── repository.py          # Usage data storage interface
│   │   │   └── service.py            # Usage tracking and aggregation logic
│   │   └── billing/                  # Invoice and payment entities
│   │       ├── __init__.py
│   │       ├── models.py
│   │       ├── proration.py          # Proration calculation logic
│   │       ├── invoice.py           # Invoice generation service
│   │       └── payment.py           # Payment processing coordination
│   └── shared/                           # Money value object and currency handling
│       ├── __init__.py
│       ├── money.py
│       ├── events.py
│       └── errors.py
└── infrastructure/                     # External system integrations and technical concerns
    ├── database/                      # DatabaseManager and transaction handling
    │   ├── __init__.py
    │   ├── connection.py
    │   ├── migrations/                 # Database schema migrations
    │   └── repositories/               # Concrete repository implementations
    │       ├── customer_repository.py
    │       ├── plan_repository.py
    │       ├── subscription_repository.py
    │       ├── usage_repository.py
    │       └── billing_repository.py
    └── payment_gateway/                # Application entry points
        └── __init__.py
```

```
    ├── client.py          # Payment gateway API client
    ├── webhook_handler.py # Webhook processing
    └── models.py          # Payment gateway data models
  └── messaging/
      ├── __init__.py
      ├── event_bus.py     # Domain event publishing and subscription
      └── queue_manager.py # Message queue integration
  └── external_services/
      ├── __init__.py
      ├── tax_service.py   # Tax calculation service integration
      └── analytics_service.py # Business intelligence integration
  └── monitoring/
      ├── __init__.py
      ├── audit_logger.py # Audit trail implementation
      └── metrics.py       # Performance and business metrics
└── api/                  # HTTP API layer
  ├── __init__.py
  ├── handlers/           # HTTP request handlers
  │   ├── plan_handlers.py
  │   ├── subscription_handlers.py
  │   ├── usage_handlers.py
  │   ├── billing_handlers.py
  │   └── webhook_handlers.py
  ├── middleware/          # HTTP middleware
  │   ├── authentication.py
  │   ├── rate_limiting.py
  │   └── error_handling.py
  └── serializers/         # Request/response serialization
    ├── plan_serializers.py
    ├── subscription_serializers.py
    └── billing_serializers.py
  └── config/              # Configuration management
    ├── __init__.py
    ├── settings.py          # BillingSettings and configuration loading
    ├── database.py          # Database configuration
    └── environments/        # Environment-specific configurations
      ├── development.py
      ├── staging.py
      └── production.py
  └── tests/                # Test suites
    ├── unit/                 # Unit tests for business logic
    │   ├── domain/
    │   └── infrastructure/
    ├── integration/          # Integration tests
    │   ├── database/
    │   ├── payment_gateway/
    │   └── api/
    ├── end_to_end/            # Full workflow tests
    │   ├── subscription_lifecycle/
    │   ├── billing_cycles/
    │   └── plan_changes/
    └── fixtures/              # Test data and mocks
      ├── customers.py
      └── plans.py
```

```

    └── usage_events.py
scripts/                      # Operational and development scripts
    ├── migrate_database.py   # Database migration runner
    ├── seed_test_data.py    # Development data seeding
    └── billing_reconciliation.py # Financial reconciliation utilities
docs/                         # Documentation
    ├── api/                 # API documentation
    ├── deployment/          # Deployment guides
    └── troubleshooting/     # Operational guides
requirements.txt               # Python dependencies
setup.py                      # Package configuration
README.md                     # Project overview and setup

```

Architecture Layer Responsibilities

The codebase structure enforces clear architectural boundaries and dependencies through its organization:

Domain Layer (`internal/domain/`)

The domain layer contains pure business logic with no external dependencies. This layer defines the core business entities, their behavior, and the rules that govern the subscription billing domain. It should be possible to test this layer without any external systems.

Directory	Responsibility	Dependencies	Key Artifacts
<code>customer/</code>	Customer entity and customer-related business rules	None (pure business logic)	Customer model, validation rules
<code>plan/</code>	Plan definitions, pricing logic, feature entitlements	Domain shared types only	Plan hierarchy, pricing calculators
<code>subscription/</code>	Subscription lifecycle, state machine, business rules	Customer and Plan domains	State machine, lifecycle events
<code>usage/</code>	Usage tracking, aggregation, quota enforcement	Subscription domain	Usage events, aggregation rules
<code>billing/</code>	Invoice generation, proration, payment coordination	All other domains	Proration algorithms, invoice logic
<code>shared/</code>	Cross-domain value objects and domain events	None	Money type, domain events, exceptions

Infrastructure Layer (`internal/infrastructure/`)

The infrastructure layer handles all external system concerns including databases, payment gateways, message queues, and monitoring systems. This layer implements the interfaces defined in the domain layer and provides concrete implementations for external integrations.

Directory	Responsibility	Dependencies	Key Artifacts
database/	Data persistence, transaction management, migrations	Domain repository interfaces	DatabaseManager , concrete repositories
payment_gateway/	Payment processing, webhook handling	Domain payment interfaces	Gateway client, webhook processor
messaging/	Event publishing, message queuing	Domain events	Event bus, queue managers
external_services/	Tax calculation, analytics, customer sync	Domain service interfaces	Service clients, data adapters
monitoring/	Audit logging, metrics, observability	Domain events and entities	Audit logger, metrics collectors

API Layer (internal/api/)

The API layer provides HTTP interfaces for external clients and handles request/response serialization, authentication, and error handling. This layer coordinates between HTTP requests and domain services but contains no business logic itself.

Directory	Responsibility	Dependencies	Key Artifacts
handlers/	HTTP request processing, response formatting	Domain services	Request handlers, response builders
middleware/	Cross-cutting HTTP concerns	Infrastructure services	Auth middleware, rate limiters
serializers/	Request/response data transformation	Domain models	Data serializers, validators

Configuration and Deployment Structure

Configuration Management (config/)

The configuration system supports multiple environments while maintaining security best practices for sensitive configuration data like database credentials and API keys.

File	Purpose	Content Type	Environment Sensitivity
<code>settings.py</code>	<code>BillingSettings</code> class and configuration loading logic	Python code	Non-sensitive defaults
<code>database.py</code>	Database connection configuration	Python code	Environment-dependent
<code>environments/development.py</code>	Development environment overrides	Python code	Development-specific settings
<code>environments/staging.py</code>	Staging environment configuration	Python code	Staging-specific settings
<code>environments/production.py</code>	Production environment configuration	Python code	Production settings (no secrets)

Testing Strategy Organization (`tests/`)

The testing structure mirrors the application architecture to ensure comprehensive coverage at all levels while maintaining test isolation and fast feedback loops.

Test Type	Directory	Purpose	Dependencies	Execution Speed
Unit Tests	<code>tests/unit/domain/</code>	Test business logic in isolation	None (mocked dependencies)	Fast (< 1s)
Integration Tests	<code>tests/integration/</code>	Test component integration	Real databases, test services	Medium (1-10s)
End-to-End Tests	<code>tests/end_to_end/</code>	Test complete workflows	Full system deployment	Slow (10s+)
Test Fixtures	<code>tests/fixtures/</code>	Shared test data and mocks	Test utilities only	N/A

Development Workflow Support

Scripts and Utilities (`scripts/`)

The scripts directory provides operational tools that developers and operators need for managing the billing system throughout its lifecycle.

Script	Purpose	Usage Context	Dependencies
<code>migrate_database.py</code>	Apply database schema migrations	Development, deployment	Database connection, migration files
<code>seed_test_data.py</code>	Create realistic test data for development	Development, testing	Domain models, repositories
<code>billing_reconciliation.py</code>	Verify billing accuracy and detect discrepancies	Operations, financial auditing	Full system access

This codebase structure provides several key benefits for development teams:

- Clear Separation of Concerns:** Business logic is isolated from infrastructure concerns, making the code easier to test and modify
- Dependency Direction:** Dependencies flow from infrastructure toward domain, never the reverse, ensuring business logic remains independent
- Test Strategy Alignment:** Test organization mirrors code organization, making it easy to find and write appropriate tests
- Scalability Support:** Structure supports adding new components or splitting services without major reorganization
- Operational Readiness:** Configuration and deployment structure supports multiple environments and operational needs

Implementation Guidance

Technology Recommendations

The following technology choices provide a solid foundation for implementing the subscription billing system while maintaining flexibility for future enhancements:

Component	Simple Option	Advanced Option	Rationale
Web Framework	Flask with Flask-RESTful	FastAPI with Pydantic	FastAPI provides automatic API documentation and better async support
Database	PostgreSQL with SQLAlchemy	PostgreSQL with async SQLAlchemy	PostgreSQL offers strong consistency guarantees for financial data
Message Queue	Redis with Python RQ	RabbitMQ with Celery	Redis provides simple setup; RabbitMQ offers better reliability guarantees
Caching	Redis (same instance as queue)	Redis Cluster	Redis handles both caching and simple queuing needs
HTTP Client	requests library	httpx with async support	httpx provides better async support for payment gateway integration
Database Migrations	Alembic (SQLAlchemy migrations)	Alembic with custom migration scripts	Alembic integrates well with SQLAlchemy ORM
Configuration	Python-decouple	Pydantic Settings	Pydantic provides type-safe configuration with validation
Testing Framework	pytest with fixtures	pytest with factories and mocks	pytest offers excellent fixture support for complex test scenarios
API Documentation	Flask-RESTX (automatic docs)	FastAPI (built-in OpenAPI)	Automatic API documentation reduces maintenance overhead

Core Infrastructure Setup

Here's the foundational infrastructure code that supports all billing components. This code handles cross-cutting concerns like database transactions, audit logging, and financial precision:

Money Value Object (`internal/domain/shared/money.py`)

```
from decimal import Decimal, ROUND_HALF_UP

from typing import Dict, Optional

from dataclasses import dataclass


# Financial precision configuration

DECIMAL_PRECISION = Decimal('0.01') # Cent precision for financial calculations

DEFAULT_CURRENCY = 'USD'


# Supported currencies with display information

SUPPORTED_CURRENCIES = {

    'USD': {'symbol': '$', 'decimal_places': 2, 'name': 'US Dollar'},

    'EUR': {'symbol': '€', 'decimal_places': 2, 'name': 'Euro'},

    'GBP': {'symbol': '£', 'decimal_places': 2, 'name': 'British Pound'},

    'JPY': {'symbol': '¥', 'decimal_places': 0, 'name': 'Japanese Yen'},

}

@dataclass(frozen=True)

class Money:

    """
    Immutable value object for financial amounts.

    Stores amounts in smallest currency unit (cents) to avoid floating point errors.
    """

    amount_cents: int

    currency_code: str


    def __post_init__(self):

        if self.currency_code not in SUPPORTED_CURRENCIES:

            raise ValueError(f"Unsupported currency: {self.currency_code}")
```

```
if self.amount_cents < 0:

    raise ValueError("Money amounts cannot be negative")




@classmethod

def from_decimal(cls, amount: Decimal, currency: str = DEFAULT_CURRENCY) -> 'Money':

    """Create Money from decimal amount (e.g., 10.50 -> Money(1050, 'USD'))"""

    currency_info = SUPPORTED_CURRENCIES[currency]

    if currency_info['decimal_places'] == 0:

        # Currencies like JPY don't have decimal places

        amount_cents = int(amount.quantize(Decimal('1'), rounding=ROUND_HALF_UP))

    else:

        # Most currencies use cents

        multiplier = Decimal(10 ** currency_info['decimal_places'])

        amount_cents = int((amount * multiplier).quantize(Decimal('1'),
rounding=ROUND_HALF_UP))

    return cls(amount_cents=amount_cents, currency_code=currency)


def to_decimal(self) -> Decimal:

    """Convert Money to decimal amount (e.g., Money(1050, 'USD') ->
Decimal('10.50'))"""

    currency_info = SUPPORTED_CURRENCIES[self.currency_code]

    if currency_info['decimal_places'] == 0:

        return Decimal(self.amount_cents)

    else:

        divisor = Decimal(10 ** currency_info['decimal_places'])

        return Decimal(self.amount_cents) / divisor
```

```
def __add__(self, other: 'Money') -> 'Money':  
  
    self._check_same_currency(other)  
  
    return Money(self.amount_cents + other.amount_cents, self.currency_code)  
  
  
def __sub__(self, other: 'Money') -> 'Money':  
  
    self._check_same_currency(other)  
  
    result_cents = self.amount_cents - other.amount_cents  
  
    if result_cents < 0:  
  
        raise ValueError("Cannot subtract to negative money amount")  
  
    return Money(result_cents, self.currency_code)  
  
  
def _check_same_currency(self, other: 'Money'):  
  
    if self.currency_code != other.currency_code:  
  
        raise ValueError(f"Cannot operate on different currencies: {self.currency_code}  
and {other.currency_code}")
```

Database Transaction Manager (`internal/infrastructure/database/connection.py`)

```
import logging

from contextlib import contextmanager

from typing import Optional, Any

from sqlalchemy import create_engine, text

from sqlalchemy.orm import sessionmaker, Session

from sqlalchemy.engine import Engine

# Database isolation levels for different operation types

ISOLATION_LEVEL_SERIALIZABLE = "SERIALIZABLE"

ISOLATION_LEVEL_READ_COMMITTED = "READ_COMMITTED"

logger = logging.getLogger(__name__)

class DatabaseManager:

    """"

    Manages database connections, transactions, and ensures proper isolation
    for financial operations that require strong consistency.

    """

    def __init__(self, connection_string: str):

        self.connection_string = connection_string

        self.engine: Engine = create_engine(

            connection_string,

            pool_pre_ping=True, # Verify connections before use

            pool_recycle=3600, # Recycle connections every hour

            echo=False # Set to True for SQL debugging

        )

        self.SessionLocal = sessionmaker(bind=self.engine)
```

PYTHON

```
@contextmanager

def transaction(self, isolation_level: str = ISOLATION_LEVEL_READ_COMMITTED):

    """
    Provide database transaction with specified isolation level.

    Use SERIALIZABLE for financial operations to prevent race conditions.
    """

    session = self.SessionLocal()

    try:

        # Set transaction isolation level
        session.execute(text(f"SET TRANSACTION ISOLATION LEVEL {isolation_level}"))

        yield session

        session.commit()

        logger.debug(f"Transaction committed with isolation level {isolation_level}")

    except Exception as e:

        session.rollback()

        logger.error(f"Transaction rolled back due to error: {str(e)}")

        raise

    finally:

        session.close()

def health_check(self) -> bool:

    """Check if database connection is healthy"""

    try:

        with self.transaction() as session:
```

```
    session.execute(text("SELECT 1"))

    return True

except Exception as e:

    logger.error(f"Database health check failed: {str(e)}")

    return False
```

Audit Event Logger (`internal/infrastructure/monitoring/audit_logger.py`)

```
import json
import logging
from datetime import datetime
from enum import Enum
from typing import Dict, Any, Optional
from dataclasses import dataclass, astuple

logger = logging.getLogger(__name__)

class AuditEventType(Enum):
    """Categories for different types of billing operations"""

    SUBSCRIPTION_CREATED = "subscription_created"
    SUBSCRIPTION_CHANGED = "subscription_changed"
    SUBSCRIPTION_CANCELLED = "subscription_cancelled"
    PLAN_CREATED = "plan_created"
    PLAN_MODIFIED = "plan_modified"
    INVOICE_GENERATED = "invoice_generated"
    PAYMENT_PROCESSED = "payment_processed"
    PAYMENT_FAILED = "payment_failed"
    USAGE_REPORTED = "usage_reported"
    PRORATION_CALCULATED = "proration_calculated"

    @dataclass
    class AuditEvent:
        """Audit trail event for billing operations"""

        event_type: AuditEventType
        entity_id: str
        entity_type: str
```

PYTHON

```
changes: Dict[str, Any]

actor_id: Optional[str]

timestamp: datetime

metadata: Dict[str, Any]

class AuditLogger:

    """
    Records audit trail for all billing operations to ensure
    compliance and enable financial reconciliation.
    """

    def __init__(self, database_manager):

        self.database_manager = database_manager


    def log_event(
        self,
        event_type: AuditEventType,
        entity_id: str,
        entity_type: str,
        changes: Dict[str, Any],
        actor_id: Optional[str] = None,
        metadata: Optional[Dict[str, Any]] = None
    ):

        """
        Record an audit event for billing operations.

        This creates an immutable record of what changed, when, and who made the change.
        """

```

```
event = AuditEvent(  
    event_type=event_type,  
    entity_id=entity_id,  
    entity_type=entity_type,  
    changes=changes,  
    actor_id=actor_id,  
    timestamp=datetime.utcnow(),  
    metadata=metadata or {}  
)  
  
try:  
    with self.database_manager.transaction() as session:  
        # Insert audit event into audit_events table  
        # Implementation would insert event data into database  
        logger.info(f"Audit event recorded: {event_type.value} for {entity_type}  
{entity_id}")  
  
except Exception as e:  
    logger.error(f"Failed to record audit event: {str(e)}")  
  
    # Audit failures should not break business operations  
  
    # but should be monitored and alerted on
```

Configuration Management (config/settings.py)

```
import os

from typing import List, Dict, Any

from dataclasses import dataclass

from decimal import Decimal


@dataclass

class BillingSettings:

    """
    Central configuration for the billing system.

    Supports environment-specific overrides while maintaining security.

    """

    # Database configuration

    database_url: str

    database_pool_size: int = 10


    # Payment gateway settings

    payment_gateway_url: str

    payment_gateway_api_key: str # Should come from environment variables

    payment_webhook_secret: str # Should come from environment variables


    # Billing behavior configuration

    default_currency: str = "USD"

    trial_period_days: int = 14

    grace_period_days: int = 7

    max_retry_attempts: int = 3
```

PYTHON

```

# Usage tracking settings

usage_aggregation_batch_size: int = 1000

usage_retention_days: int = 365


# Financial precision

rounding_precision: Decimal = Decimal('0.01')


# External service timeouts

payment_gateway_timeout: int = 30 # seconds

tax_service_timeout: int = 10      # seconds


@classmethod

def from_environment(cls) -> 'BillingSettings':
    """Load configuration from environment variables with validation"""

    return cls(
        database_url=os.getenv('DATABASE_URL', 'postgresql://localhost/billing'),
        database_pool_size=int(os.getenv('DATABASE_POOL_SIZE', '10')),
        payment_gateway_url=os.getenv('PAYMENT_GATEWAY_URL', 'https://api.stripe.com'),
        payment_gateway_api_key=os.getenv('PAYMENT_GATEWAY_API_KEY', ''),
        payment_webhook_secret=os.getenv('PAYMENT_WEBHOOK_SECRET', ''),
        trial_period_days=int(os.getenv('TRIAL_PERIOD_DAYS', '14')),
        grace_period_days=int(os.getenv('GRACE_PERIOD_DAYS', '7')),
    )

```

Component Service Skeletons

Here are the core service interfaces that implement the main billing components. These provide the structure for implementing each milestone:

Plan Management Service (`internal/domain/plan/service.py`)

```
from typing import List, Optional
from .models import Plan, PlanVersion
from .repository import PlanRepository

class PlanManagementService:

    """
    Handles subscription plan creation, modification, and versioning.
    Corresponds to Milestone 1: Plans & Pricing.
    """

    def __init__(self, plan_repository: PlanRepository, audit_logger):
        self.plan_repository = plan_repository
        self.audit_logger = audit_logger

    def create_plan(self, plan_data: dict) -> Plan:
        """
        Create new subscription plan with pricing and feature configuration.

        TODO 1: Validate plan data structure and pricing configuration
        TODO 2: Check for plan name uniqueness within organization
        TODO 3: Create Plan entity with initial version
        TODO 4: Validate pricing tiers and feature entitlements
        TODO 5: Persist plan and log audit event
        TODO 6: Return created plan with generated ID
        """
        pass
```

PYTHON

```
def modify_plan(self, plan_id: str, changes: dict) -> PlanVersion:
    """
    Create new version of existing plan while preserving existing subscriptions.

    TODO 1: Load existing plan and validate modification permissions
    TODO 2: Create new plan version with changes applied
    TODO 3: Validate new version maintains backward compatibility
    TODO 4: Archive old version and activate new version
    TODO 5: Log plan modification audit event
    TODO 6: Return new plan version
    """
    pass

def calculate_plan_price(self, plan_id: str, quantity: int = 1) -> 'Money':
    """
    Calculate pricing for plan based on quantity and pricing model.

    TODO 1: Load plan pricing configuration
    TODO 2: Determine pricing model (flat, per-seat, tiered, volume)
    TODO 3: Apply quantity to pricing calculation
    TODO 4: Handle tiered vs volume pricing correctly
    TODO 5: Return Money object with calculated amount
    """
    pass
```

Subscription Lifecycle Service ([internal/domain/subscription/service.py](#))

```
from typing import Optional
```

PYTHON

```
from datetime import datetime
```

```
from .models import Subscription, SubscriptionState
```

```
from .repository import SubscriptionRepository
```

```
class SubscriptionLifecycleService:
```

```
    """
```

```
    Manages complete subscription lifecycle from creation to cancellation.
```

```
    Corresponds to Milestone 2: Subscription Lifecycle.
```

```
    """
```

```
def __init__(self, subscription_repository: SubscriptionRepository,
```

```
            plan_service, audit_logger):
```

```
    self.subscription_repository = subscription_repository
```

```
    self.plan_service = plan_service
```

```
    self.audit_logger = audit_logger
```

```
def create_subscription(self, customer_id: str, plan_id: str,
```

```
                      payment_method_id: str, trial_end: Optional[datetime] = None) ->
```

```
Subscription:
```

```
    """
```

```
    Create new subscription with trial or immediate activation.
```

```
    TODO 1: Validate customer exists and has valid payment method
```

```
    TODO 2: Load and validate plan is available for new subscriptions
```

```
    TODO 3: Calculate billing anchor date and first billing date
```

```
    TODO 4: Create subscription entity with appropriate initial state
```

```
    TODO 5: Set up trial period if specified or plan includes trial
```

```
    TODO 6: Persist subscription and publish SubscriptionCreated event
```

```
    TODO 7: Return created subscription with all details
```

```
    """
```

```
    pass
```

```
def process_renewal(self, subscription_id: str) -> bool:
```

```
    """
```

```
    Process subscription renewal at end of billing period.
```

```
    TODO 1: Load subscription and verify it's due for renewal
```

```
    TODO 2: Check subscription is in renewable state (active, past_due)
```

```
    TODO 3: Generate invoice for upcoming billing period
```

```
    TODO 4: Attempt payment processing through payment service
```

```
    TODO 5: Update subscription state based on payment result
```

```
    TODO 6: Schedule next renewal date if payment succeeded
```

```
    TODO 7: Return success/failure status
```

```
    """
```

```
    pass
```

```
def cancel_subscription(self, subscription_id: str,
```

```
                        cancel_at_period_end: bool = True, reason: str = None) ->
```

```
Subscription:
```

```
    """
```

```
    Cancel subscription immediately or at end of current billing period.
```

```
    TODO 1: Load subscription and validate it can be cancelled
```

```
    TODO 2: Calculate effective cancellation date based on cancel_at_period_end
```

```
TODO 3: Update subscription state to cancelled or scheduled for cancellation

TODO 4: Process any refunds if cancelling mid-period

TODO 5: Clean up recurring payment schedules

TODO 6: Log cancellation audit event with reason

TODO 7: Return updated subscription

"""

pass
```

File Structure Implementation

Here's how to organize the initial project structure with key files:

Project Initialization Script (`setup_project.py`)

```
#!/usr/bin/env python3
```

PYTHON

```
"""
```

```
Script to initialize the subscription billing system project structure.
```

```
Run this to create all necessary directories and placeholder files.
```

```
"""
```

```
import os
```

```
from pathlib import Path
```

```
def create_directory_structure():
```

```
    """Create the complete project directory structure"""

    directories = [
```

```
        'cmd/billing-server',
        'cmd/usage-ingestion',
        'cmd/billing-scheduler',
        'internal/domain/customer',
        'internal/domain/plan',
        'internal/domain/subscription',
        'internal/domain/usage',
        'internal/domain/billing',
        'internal/domain/shared',
        'internal/infrastructure/database',
        'internal/infrastructure/database/repositories',
        'internal/infrastructure/database/migrations',
        'internal/infrastructure/payment_gateway',
        'internal/infrastructure/messaging',
        'internal/infrastructure/external_services',
```

```

'internal/infrastructure/monitoring',
'internal/api/handlers',
'internal/api/middleware',
'internal/api/serializers',
'config/environments',
'tests/unit/domain',
'tests/integration',
'tests/end_to_end',
'tests/fixtures',
'scripts',
'docs/api',
'docs/deployment',
'docs/troubleshooting'

]

for directory in directories:
    Path(directory).mkdir(parents=True, exist_ok=True)

    # Create __init__.py files for Python packages

    if 'internal' in directory or 'config' in directory or 'tests' in directory:
        (Path(directory) / '__init__.py').touch()

print("✅ Project directory structure created successfully")

if __name__ == '__main__':
    create_directory_structure()

```

Milestone Validation Checkpoints

After implementing each component, use these checkpoints to verify functionality:

Milestone 1 Checkpoint: Plan Management

```
# Test plan creation and pricing calculation  
  
python -m pytest tests/unit/domain/plan/ -v  
  
python -c "  
  
from internal.domain.plan.service import PlanManagementService  
  
# Verify plan creation works with sample data  
  
print('✅ Plan management component ready')  
  
"
```

BASH

Milestone 2 Checkpoint: Subscription Lifecycle

```
# Test subscription state machine and lifecycle  
  
python -m pytest tests/unit/domain/subscription/ -v  
  
python -c "  
  
from internal.domain.subscription.service import SubscriptionLifecycleService  
  
# Verify subscription creation and state transitions  
  
print('✅ Subscription lifecycle component ready')  
  
"
```

BASH

Integration Checkpoint: Database and Payment Gateway

```
# Test database connectivity and payment integration

python -c "
from config.settings import BillingSettings
from internal.infrastructure.database.connection import DatabaseManager

settings = BillingSettings.from_environment()

db = DatabaseManager(settings.database_url)

if db.health_check():

    print('✅ Database connection working')

else:

    print('❌ Database connection failed')

"
"
```

BASH

This implementation guidance provides the foundation for building the subscription billing system with proper separation of concerns, financial precision, and scalability. The structure supports both current requirements and future enhancements while maintaining clean architectural boundaries.

Data Model and Core Entities

Milestone(s): Foundation for all milestones - provides the data structures and relationships that support plan management (Milestone 1), subscription lifecycle (Milestone 2), proration calculations (Milestone 3), and usage-based billing (Milestone 4)

Mental Model: The Financial Ledger System

Think of the subscription billing data model like a traditional accounting ledger system that a bank might use. Each customer is like an account holder with a unique account number. Subscription plans are like different types of banking products (checking accounts, savings accounts, credit cards) - each with specific terms, fees, and features. Active subscriptions are like opened accounts that generate recurring transactions. Invoices are like monthly statements that itemize all charges and payments. Usage events are like individual transaction records that get aggregated into summary line items on the statement.

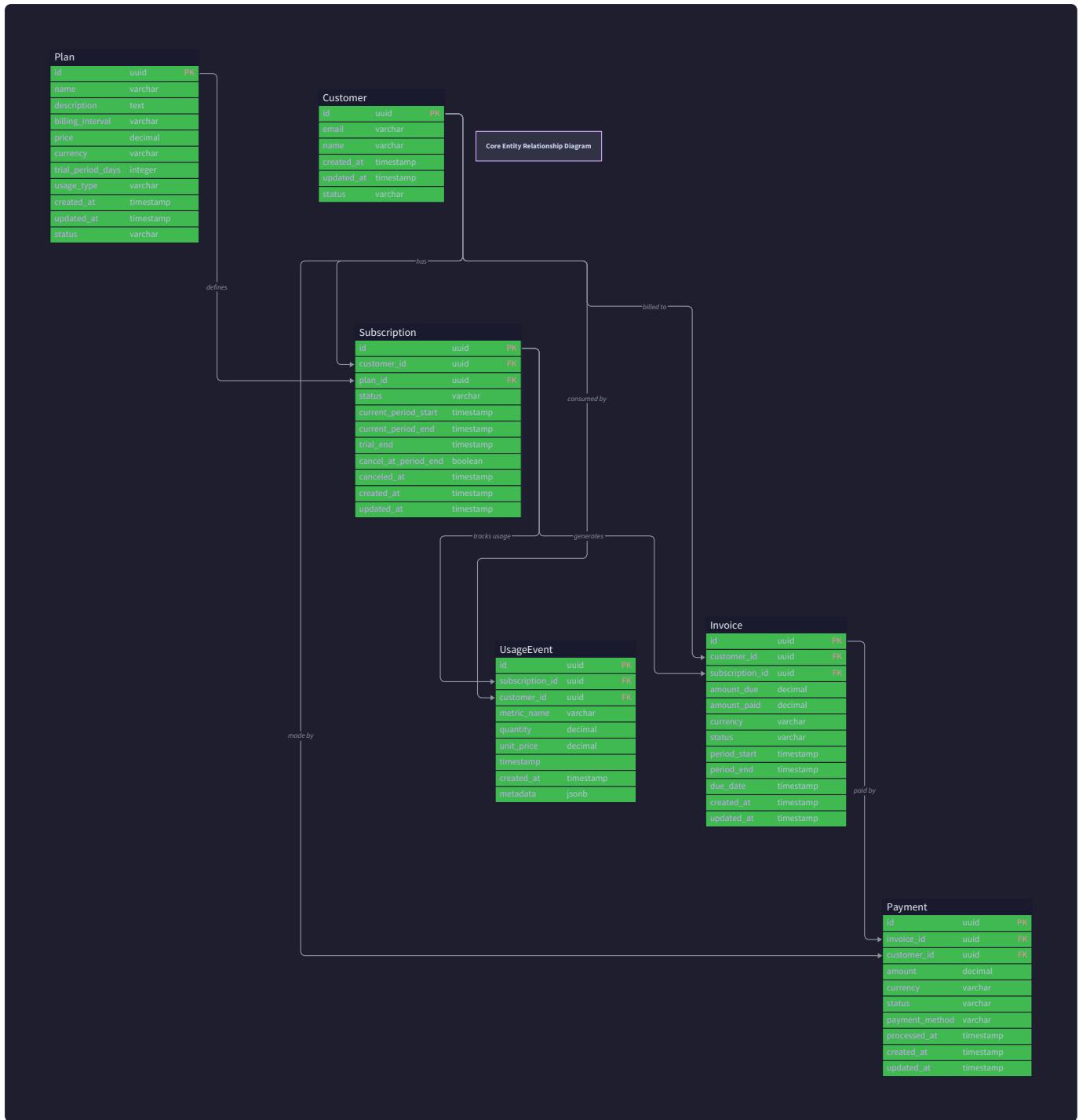
Just as a bank must maintain perfect accuracy in financial records and preserve complete audit trails, our billing system must ensure every monetary amount is precise to the cent, every state change is tracked, and

every calculation can be verified. The data model serves as the foundation for financial integrity - any errors or inconsistencies at this level cascade into billing disputes, revenue leakage, and compliance violations.

The key insight is that subscription billing is fundamentally an event-sourced financial system. Every significant business event (subscription creation, plan changes, usage consumption, payments) generates immutable records that collectively define the current state. This approach ensures we can always reconstruct how we arrived at any billing amount and provides the auditability that financial systems require.

Core Business Entities

The subscription billing system is built around five core entities that represent the fundamental business concepts. Each entity captures specific aspects of the customer relationship and billing lifecycle, with carefully designed relationships that maintain referential integrity and support complex billing scenarios.



Customer Entity

The `Customer` entity represents the billing account holder and serves as the root aggregate for all subscription-related activities. A customer can have multiple subscriptions but maintains unified billing settings and credit balances across all their services.

Field Name	Type	Description
<code>customer_id</code>	UUID	Primary key, immutable unique identifier for the customer
<code>external_id</code>	String	Optional external system identifier for integration purposes
<code>email</code>	String	Primary email address for billing communications and account recovery
<code>name</code>	String	Full name or company name for invoice display
<code>billing_address</code>	JSON	Structured address object with street, city, state, postal_code, country
<code>tax_id</code>	String	Tax identification number for tax calculation and compliance
<code>currency</code>	String	Preferred currency code (ISO 4217) for all billing in this account
<code>credit_balance_cents</code>	Integer	Current credit balance in smallest currency unit (cents)
<code>payment_method_id</code>	UUID	Reference to default payment method in payment gateway
<code>billing_email</code>	String	Optional separate email for billing notifications
<code>account_status</code>	Enum	Current account status: active, suspended, closed
<code>created_at</code>	Timestamp	Account creation date for analytics and lifecycle tracking
<code>updated_at</code>	Timestamp	Last modification timestamp for change tracking
<code>metadata</code>	JSON	Extensible field for custom attributes and integration data

The customer entity includes a `credit_balance_cents` field that accumulates credits from downgrades, refunds, and promotional credits. This balance is automatically applied to future invoices, providing a seamless experience for customers receiving credits. The `billing_address` and `tax_id` fields support tax calculation requirements and compliance with regional billing regulations.

Plan Entity

The `Plan` entity defines the pricing structure, billing terms, and feature entitlements for subscription offerings. Plans are versioned to ensure existing subscribers maintain their original terms while allowing the business to evolve pricing over time.

Field Name	Type	Description
<code>plan_id</code>	UUID	Primary key, immutable unique identifier for this plan version
<code>plan_code</code>	String	Human-readable identifier (e.g., "pro-monthly-v2")
<code>name</code>	String	Display name for customer-facing interfaces
<code>description</code>	String	Detailed description of plan features and benefits
<code>status</code>	Enum	Plan status: active, deprecated, archived
<code>billing_interval</code>	Enum	Recurring billing period: monthly, quarterly, annual
<code>billing_interval_count</code>	Integer	Number of intervals between charges (e.g., 2 for bi-monthly)
<code>pricing_model</code>	Enum	Pricing structure: flat_rate, per_seat, tiered, volume
<code>base_price_cents</code>	Integer	Base recurring charge in smallest currency unit
<code>currency</code>	String	Currency code (ISO 4217) for all pricing in this plan
<code>trial_period_days</code>	Integer	Length of free trial period, null if no trial offered
<code>setup_fee_cents</code>	Integer	One-time setup charge applied at subscription creation
<code>features</code>	JSON	Array of feature flags and entitlements included in this plan
<code>usage_limits</code>	JSON	Monthly usage quotas for metered features
<code>pricing_tiers</code>	JSON	Tiered pricing structure for usage-based billing
<code>created_at</code>	Timestamp	Plan creation date for lifecycle tracking
<code>updated_at</code>	Timestamp	Last modification timestamp
<code>deprecated_at</code>	Timestamp	Date when plan was deprecated (null if still active)
<code>metadata</code>	JSON	Extensible field for custom plan attributes

The `pricing_tiers` field supports complex usage-based billing scenarios. For example, an API plan might have tiers like: "First 1,000 calls free, next 9,000 calls at \$0.001 each, additional calls at \$0.0005 each." The `features` field contains an array of feature flags that the entitlement system uses to control access to application functionality.

Decision: Plan Versioning Strategy

- **Context:** When businesses need to change pricing or features, they must decide whether to update existing plans or create new versions
- **Options Considered:**
 1. Update plans in-place (simple but breaks existing customer terms)
 2. Create new plan versions and deprecate old ones
 3. Full plan history with temporal validity periods
- **Decision:** Create new plan versions and deprecate old ones
- **Rationale:** Preserves existing customer terms (critical for trust), provides clear upgrade paths, and maintains audit trail while avoiding the complexity of temporal data management
- **Consequences:** Requires plan migration workflows and slightly more complex plan selection logic, but ensures customer contract integrity

Subscription Entity

The `Subscription` entity represents an active customer enrollment in a specific plan. It tracks the subscription lifecycle, billing schedule, and current state while maintaining complete history of plan changes and modifications.

Field Name	Type	Description
<code>subscription_id</code>	UUID	Primary key, immutable unique identifier
<code>customer_id</code>	UUID	Foreign key reference to customer entity
<code>plan_id</code>	UUID	Foreign key reference to current plan
<code>status</code>	Enum	Current state: trial, active, past_due, cancelled, expired
<code>billing_cycle_anchor</code>	Integer	Day of month for recurring billing (1-31)
<code>current_period_start</code>	Date	Start date of current billing period
<code>current_period_end</code>	Date	End date of current billing period
<code>trial_start</code>	Date	Trial period start date (null if no trial)
<code>trial_end</code>	Date	Trial period end date (null if no trial)
<code>cancelled_at</code>	Timestamp	Cancellation request timestamp (null if not cancelled)
<code>cancel_at_period_end</code>	Boolean	Whether cancellation takes effect at period end
<code>ended_at</code>	Timestamp	Actual subscription termination timestamp
<code>quantity</code>	Integer	Number of seats or units for per-seat pricing
<code>proration_behavior</code>	Enum	How to handle mid-cycle changes: immediate, next_cycle
<code>collection_method</code>	Enum	Payment timing: charge Automatically, send_invoice
<code>days_until_due</code>	Integer	Payment terms for invoice-based collection
<code>created_at</code>	Timestamp	Subscription creation timestamp
<code>updated_at</code>	Timestamp	Last modification timestamp
<code>metadata</code>	JSON	Custom attributes and integration data

The `billing_cycle_anchor` field ensures consistent billing dates. For example, a subscription created on January 31st with a monthly billing interval will bill on the last day of each month (February 28th, March 31st, etc.) rather than causing month overflow issues. The `proration_behavior` setting controls whether plan changes take effect immediately with prorated charges or wait until the next billing cycle.

Invoice Entity

The `Invoice` entity represents a billing statement that itemizes charges, credits, and payment details for a specific billing period. Invoices are immutable once finalized to ensure financial accuracy and audit compliance.

Field Name	Type	Description
invoice_id	UUID	Primary key, immutable unique identifier
customer_id	UUID	Foreign key reference to customer
subscription_id	UUID	Primary subscription for this invoice (nullable for one-off charges)
invoice_number	String	Sequential human-readable identifier (e.g., "INV-2024-001234")
status	Enum	Invoice state: draft, open, paid, void, uncollectible
period_start	Date	Billing period start date
period_end	Date	Billing period end date
subtotal_cents	Integer	Total charges before credits and taxes
credit_applied_cents	Integer	Credits applied from customer balance
tax_cents	Integer	Tax amount calculated based on customer location
total_cents	Integer	Final amount due after credits and taxes
amount_paid_cents	Integer	Amount actually collected
amount_remaining_cents	Integer	Outstanding balance (total - paid)
currency	String	Currency code for all amounts on this invoice
due_date	Date	Payment due date for collection
paid_at	Timestamp	Payment completion timestamp (null if unpaid)
voided_at	Timestamp	Void timestamp (null if not voided)
payment_attempt_count	Integer	Number of payment collection attempts
next_payment_attempt	Timestamp	Scheduled retry timestamp for failed payments
created_at	Timestamp	Invoice generation timestamp
finalized_at	Timestamp	When invoice became immutable
metadata	JSON	Additional invoice attributes

The invoice tracks separate amounts for charges, credits, taxes, and payments to provide complete transparency in billing calculations. The `payment_attempt_count` and `next_payment_attempt` fields support dunning management by tracking retry attempts and scheduling future collection efforts.

Invoice Line Item Entity

Invoice line items provide detailed breakdown of charges and credits on each invoice. This granular structure supports complex billing scenarios including proration, usage charges, and plan changes.

Field Name	Type	Description
line_item_id	UUID	Primary key, immutable unique identifier
invoice_id	UUID	Foreign key reference to parent invoice
subscription_id	UUID	Subscription generating this line item (nullable)
type	Enum	Line item type: subscription, usage, proration_credit, setup_fee
description	String	Human-readable description of the charge
period_start	Date	Service period start date for this line item
period_end	Date	Service period end date for this line item
quantity	Decimal	Units being charged (e.g., seats, API calls)
unit_amount_cents	Integer	Price per unit in smallest currency unit
amount_cents	Integer	Total amount for this line item (quantity × unit_amount)
proration_factor	Decimal	Percentage of billing period (e.g., 0.5 for half month)
plan_id	UUID	Plan associated with this charge (nullable)
usage_summary	JSON	Aggregated usage data for usage-based line items
created_at	Timestamp	Line item creation timestamp
metadata	JSON	Additional line item attributes

The `proration_factor` field captures the exact percentage of a billing period that a charge covers, enabling precise proration calculations. For example, a mid-month upgrade from a \$10 plan to a \$20 plan with 15 days remaining would generate a proration credit line item with factor 0.5 (15/30 days) for -\$5.

Payment Entity

The `Payment` entity records payment attempts and their results, integrating with the external payment gateway while maintaining local records for reconciliation and reporting.

Field Name	Type	Description
<code>payment_id</code>	UUID	Primary key, immutable unique identifier
<code>customer_id</code>	UUID	Foreign key reference to customer
<code>invoice_id</code>	UUID	Foreign key reference to invoice being paid
<code>payment_intent_id</code>	String	External payment gateway transaction identifier
<code>amount_cents</code>	Integer	Payment amount in smallest currency unit
<code>currency</code>	String	Payment currency code
<code>status</code>	Enum	Payment state: pending, succeeded, failed, cancelled, refunded
<code>payment_method_type</code>	String	Payment method used (card, bank_transfer, etc.)
<code>failure_code</code>	String	Error code from payment gateway (null if successful)
<code>failure_message</code>	String	Human-readable failure description
<code>gateway_response</code>	JSON	Complete response from payment processor
<code>captured_at</code>	Timestamp	When payment was captured (null if pending)
<code>refunded_at</code>	Timestamp	Refund timestamp (null if not refunded)
<code>refunded_amount_cents</code>	Integer	Amount refunded (null if no refund)
<code>created_at</code>	Timestamp	Payment attempt timestamp
<code>metadata</code>	JSON	Additional payment attributes

Entity Relationships and Constraints

The relationships between entities enforce business rules and maintain data integrity across the subscription lifecycle. These constraints prevent invalid states and ensure consistent billing behavior.

Primary Relationships

The core entity relationships follow a hierarchical structure where customers own subscriptions, subscriptions generate invoices, and invoices are settled by payments. This design supports both simple single-subscription customers and complex enterprise accounts with multiple subscriptions and consolidated billing.

Relationship	Cardinality	Foreign Key	Constraint Description
Customer → Subscription	One to Many	<code>subscription.customer_id</code>	Each customer can have multiple active subscriptions
Plan → Subscription	One to Many	<code>subscription.plan_id</code>	Multiple subscriptions can use the same plan
Customer → Invoice	One to Many	<code>invoice.customer_id</code>	All invoices belong to a specific customer
Subscription → Invoice	One to Many	<code>invoice.subscription_id</code>	Subscriptions generate recurring invoices
Invoice → Line Item	One to Many	<code>line_item.invoice_id</code>	Each invoice contains one or more line items
Invoice → Payment	One to Many	<code>payment.invoice_id</code>	Invoices may require multiple payment attempts

Decision: Subscription-Invoice Relationship

- **Context:** Some billing systems link invoices directly to billing periods, others to specific subscriptions
- **Options Considered:**
 1. Link invoices to subscriptions (enables subscription-specific billing)
 2. Link invoices to customers only (enables consolidated billing)
 3. Support both models with nullable subscription references
- **Decision:** Support both models with nullable subscription references
- **Rationale:** Provides flexibility for enterprise customers who need consolidated billing while maintaining subscription-specific invoices for simpler use cases
- **Consequences:** Requires conditional logic in billing workflows but supports both B2B and B2C billing patterns

Business Rule Constraints

The subscription billing system enforces several critical business rules through database constraints and application logic. These rules prevent data corruption and ensure billing accuracy.

Constraint	Entity	Rule Description	Enforcement Method
Currency Consistency	Customer, Plan, Subscription	All entities in a billing relationship must use the same currency	Application validation
Billing Period Validity	Subscription	<code>current_period_end</code> must be after <code>current_period_start</code>	Database check constraint
Trial Period Logic	Subscription	Trial end must be before or equal to first billing date	Application validation
Invoice Immutability	Invoice	Finalized invoices cannot be modified	Application logic
Payment Amount Validation	Payment	Payment amount cannot exceed invoice total	Application validation
Status Transitions	Subscription, Invoice	Only valid state transitions are allowed	Application state machine
Credit Balance Consistency	Customer	Credit balance cannot be negative after applying credits	Database check constraint
Plan Version Integrity	Plan	Deprecated plans cannot be assigned to new subscriptions	Application validation

The currency consistency constraint prevents billing errors that could occur when mixing currencies within a customer account. For example, a customer with USD billing cannot be assigned a plan priced in EUR without explicit currency conversion.

Data Integrity Rules

Financial systems require additional integrity safeguards beyond standard relational constraints. These rules ensure monetary calculations are accurate and auditable.

Money Amount Precision: All monetary amounts are stored as integers representing the smallest currency unit (cents for USD). This eliminates floating-point precision errors that could accumulate across many billing cycles. The system uses a `Money` type that encapsulates amount and currency to prevent mixing currencies in calculations.

Idempotent Operations: Critical operations like invoice generation and payment processing must be idempotent to prevent duplicate charges. Each operation includes an idempotency key that prevents duplicate execution if requests are retried.

Audit Trail Requirements: Every modification to subscription state, plan assignments, and billing amounts generates an audit event. This provides complete traceability for compliance and customer service purposes.

Referential Integrity: Foreign key relationships are enforced at the database level where possible, with application-level validation for complex business rules. Cascade delete operations are restricted to prevent accidental data loss.

Usage Tracking Data Model

Usage-based billing requires a separate data model optimized for high-volume event ingestion and efficient aggregation. This model supports real-time usage tracking while providing the performance needed for billing calculations across large customer bases.

Usage Event Entity

The `UsageEvent` entity captures individual metered activities that contribute to usage-based charges. Events are designed to be immutable and idempotent to ensure accurate billing regardless of retry behavior or duplicate submissions.

Field Name	Type	Description
<code>event_id</code>	UUID	Primary key, immutable unique identifier
<code>idempotency_key</code>	String	Client-provided key for duplicate detection
<code>customer_id</code>	UUID	Foreign key reference to customer
<code>subscription_id</code>	UUID	Foreign key reference to subscription
<code>event_type</code>	String	Meter name (e.g., "api_calls", "storage_gb")
<code>quantity</code>	Decimal	Amount consumed in this event
<code>unit</code>	String	Unit of measurement (calls, GB, hours)
<code>timestamp</code>	Timestamp	When the usage occurred (client-reported)
<code>processed_at</code>	Timestamp	When the event was received and validated
<code>billing_period</code>	String	Billing period this event contributes to (YYYY-MM format)
<code>properties</code>	JSON	Additional event metadata and dimensions
<code>source_system</code>	String	System that generated this usage event
<code>aggregated</code>	Boolean	Whether this event has been included in billing calculations
<code>created_at</code>	Timestamp	Event record creation timestamp

The `idempotency_key` field prevents double-counting when client systems retry usage submissions. The billing system checks for existing events with the same idempotency key and customer before creating new

records. The `billing_period` field enables efficient querying for aggregation without complex date range calculations.

Decision: Usage Event Aggregation Strategy

- **Context:** Usage events can accumulate to millions of records per month, requiring efficient aggregation for billing
- **Options Considered:**
 1. Real-time aggregation into running totals (fast billing but complex consistency)
 2. Batch aggregation during billing cycle (simple but potential delays)
 3. Hybrid approach with periodic pre-aggregation and billing-time finalization
- **Decision:** Hybrid approach with periodic pre-aggregation
- **Rationale:** Balances billing speed with system complexity while providing near real-time usage visibility for customers
- **Consequences:** Requires background aggregation jobs and eventual consistency handling, but enables responsive usage dashboards and fast billing

Usage Aggregation Entity

The `UsageAggregation` entity stores pre-computed usage totals for efficient billing calculations.

Aggregations are updated periodically and finalized during billing to ensure accuracy.

Field Name	Type	Description
aggregation_id	UUID	Primary key, immutable unique identifier
customer_id	UUID	Foreign key reference to customer
subscription_id	UUID	Foreign key reference to subscription
event_type	String	Meter name being aggregated
billing_period	String	Billing period for this aggregation (YYYY-MM format)
period_start	Date	Start date of the billing period
period_end	Date	End date of the billing period
total_quantity	Decimal	Total usage amount for this period
event_count	Integer	Number of events included in aggregation
last_event_timestamp	Timestamp	Timestamp of most recent event included
is_finalized	Boolean	Whether aggregation is complete for billing
finalized_at	Timestamp	When aggregation was finalized
updated_at	Timestamp	Last aggregation update timestamp

Usage Billing Entity

The `UsageBilling` entity calculates billable amounts from usage aggregations based on plan pricing tiers and limits. This separation allows for complex usage billing scenarios and provides transparency in billing calculations.

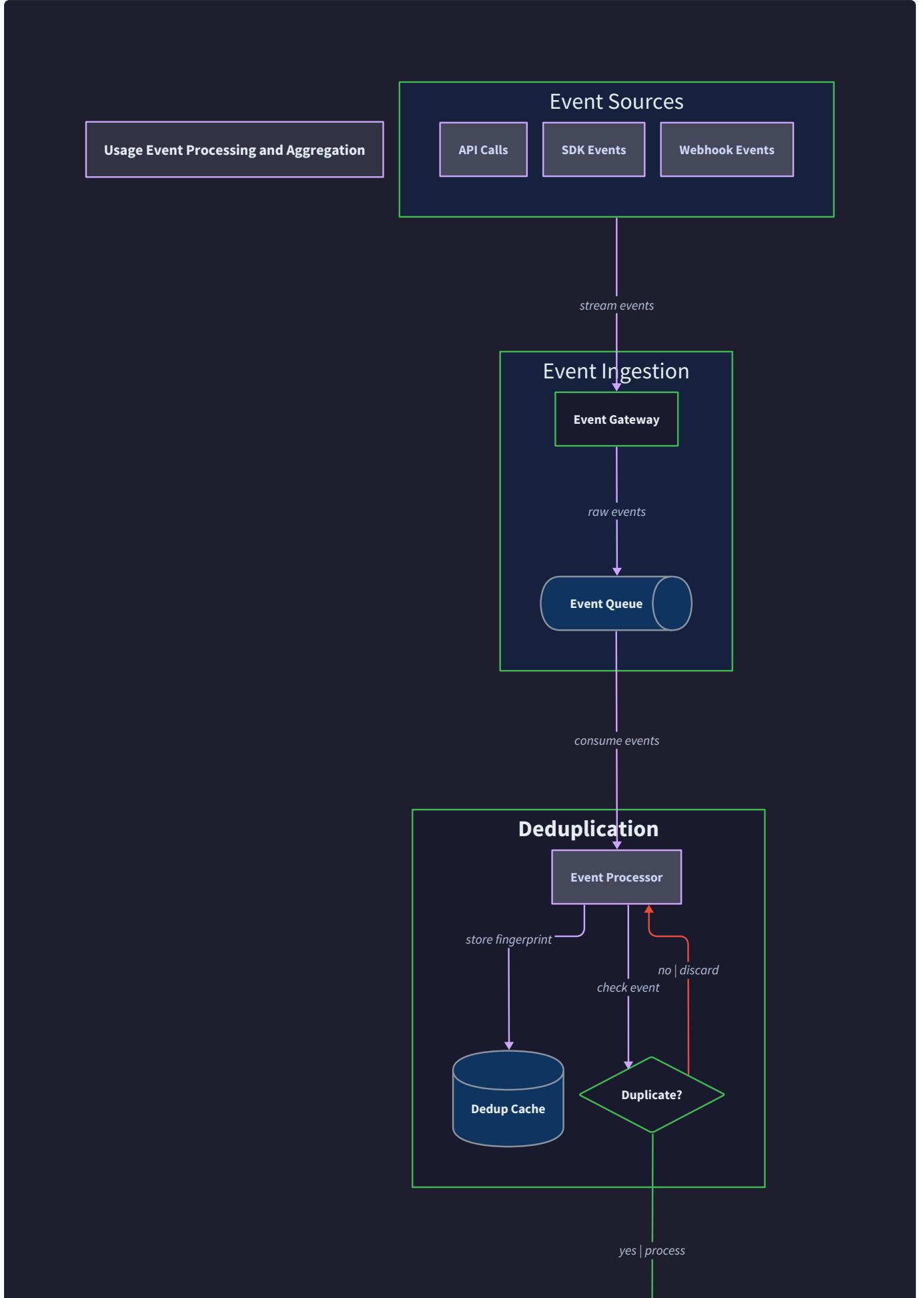
Field Name	Type	Description
usage_billing_id	UUID	Primary key, immutable unique identifier
aggregation_id	UUID	Foreign key reference to usage aggregation
plan_id	UUID	Plan used for billing calculation
event_type	String	Meter name being billed
billing_period	String	Billing period for this calculation
included_quantity	Decimal	Usage amount included in plan (free tier)
billable_quantity	Decimal	Usage amount subject to charges
total_amount_cents	Integer	Total charges for this usage type
tier_breakdown	JSON	Detailed calculation showing each pricing tier
calculated_at	Timestamp	When billing calculation was performed
invoice_line_item_id	UUID	Reference to generated invoice line item

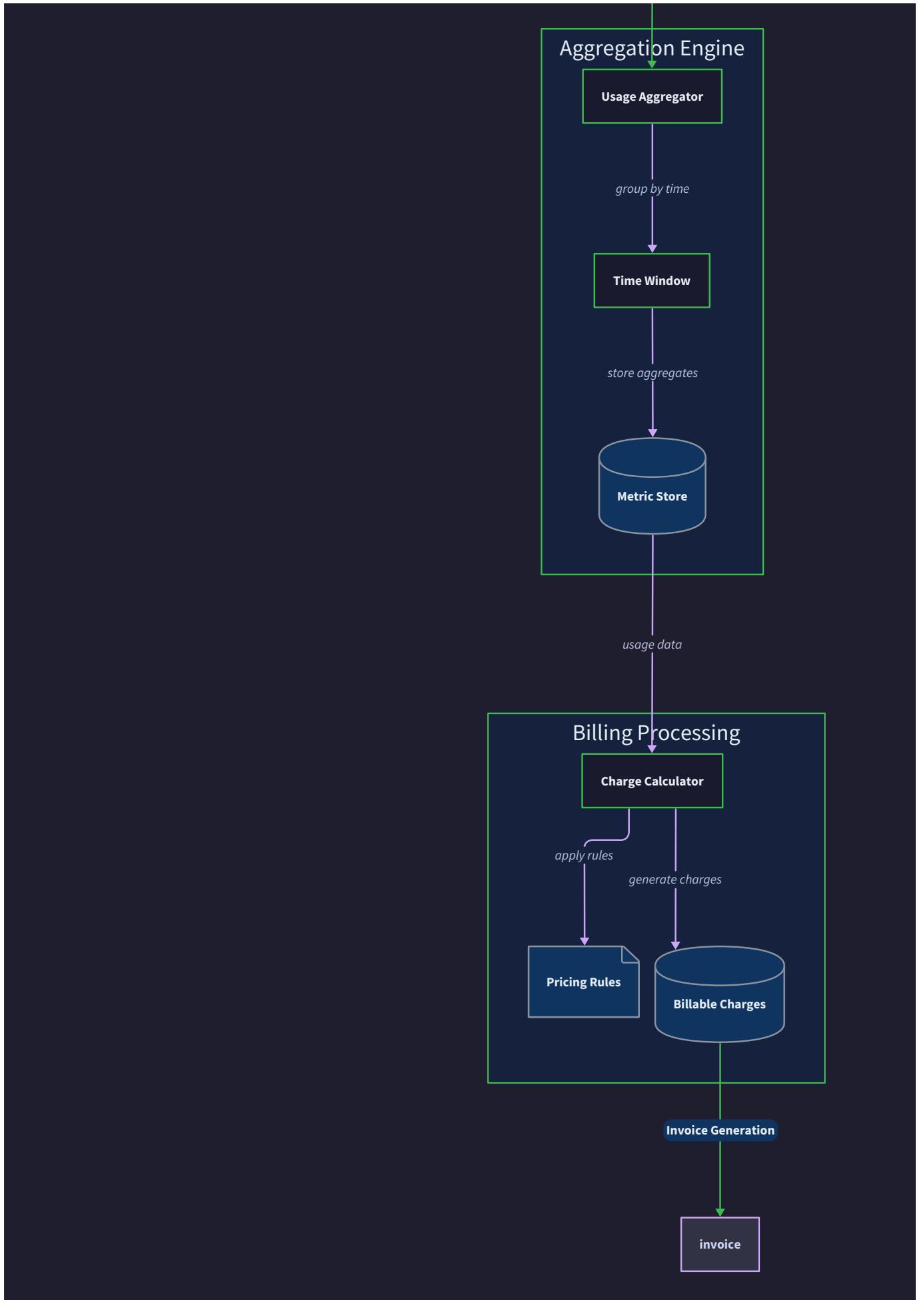
The `tier_breakdown` field contains a detailed calculation showing how usage was allocated across pricing tiers. For example:

```
{"tiers": [{"range": "0-1000", "quantity": 1000, "rate_cents": 0, "amount_cents": 0}, {"range": "1001-5000", "quantity": 2500, "rate_cents": 1, "amount_cents": 2500}]}}
```

Usage Event Processing Flow

Usage events flow through several processing stages to ensure accuracy and performance in high-volume scenarios. This pipeline architecture supports real-time ingestion while maintaining billing accuracy.





- 1. Event Ingestion:** Client systems submit usage events via API with idempotency keys. Events are validated for required fields, reasonable quantities, and timestamp constraints.
- 2. Deduplication:** The system checks `idempotency_key` and `customer_id` combinations to prevent duplicate events. Duplicate submissions return the original event ID without creating new records.
- 3. Temporal Assignment:** Events are assigned to billing periods based on the customer's subscription billing cycle, not calendar months. This ensures usage aligns with subscription billing boundaries.
- 4. Aggregation:** Background processes periodically aggregate events into usage totals, updating the `aggregated` flag to prevent double-counting in future runs.
- 5. Billing Calculation:** During invoice generation, usage aggregations are converted to billable amounts using current plan pricing. This separation allows for plan changes without re-processing historical events.
- 6. Audit and Reconciliation:** Usage totals are validated against event counts and quantities to detect processing errors or data corruption.

Common Usage Tracking Pitfalls

⚠ Pitfall: Clock Skew in Event Timestamps When client systems report usage events with their local timestamps, clock differences can cause events to appear in the wrong billing period. This leads to usage being credited to the wrong invoice.

Why it's wrong: A client system with a fast clock might report usage events with timestamps in the next billing period, causing them to be excluded from the current invoice.

How to fix: Use server-side processing timestamps for billing period assignment while preserving client timestamps for audit purposes. Implement reasonable timestamp validation to reject events with timestamps too far in the future or past.

⚠ Pitfall: Aggregation Boundary Errors Incorrect billing period boundaries can cause usage to be split across multiple invoices or excluded entirely. This is especially problematic for subscriptions with non-standard billing anchors.

Why it's wrong: Using calendar month boundaries instead of subscription-specific billing periods causes usage to be prorated incorrectly or appear on the wrong invoice.

How to fix: Calculate billing periods using the subscription's `billing_cycle_anchor` and `current_period_start / current_period_end` dates. Store the computed billing period with each event to avoid recalculation errors.

⚠ Pitfall: Usage Tier Calculation Errors Complex pricing tiers can be implemented incorrectly, leading to overcharges or undercharges. Tiered pricing (each tier priced separately) is often confused with volume pricing (single rate based on total usage).

Why it's wrong: Applying a single rate to total usage instead of calculating each tier separately can dramatically overcharge customers in higher usage tiers.

How to fix: Implement tier calculations that iterate through each pricing tier, calculating charges for usage within that tier's range, and sum the results. Store the tier breakdown for transparency and debugging.

Implementation Guidance

This section provides the foundational code structure and starter implementations for the core data model. The focus is on creating robust, type-safe implementations that prevent common billing errors.

Technology Recommendations

Component	Simple Option	Advanced Option
Database	PostgreSQL with SQLAlchemy ORM	PostgreSQL with async SQLAlchemy + connection pooling
Money Handling	Custom Money class with Decimal	Money library (py-money) with currency validation
UUID Generation	uuid.uuid4()	shortuuid for human-readable IDs
Timestamp Handling	datetime with UTC timezone	pendulum library for robust datetime operations
JSON Validation	Manual validation	Pydantic models for structured JSON fields
Database Migrations	Alembic (included with SQLAlchemy)	Alembic with custom migration templates

Recommended File Structure

```
subscription-billing/
├── src/
│   ├── models/
│   │   ├── __init__.py           ← Export all entities
│   │   ├── base.py              ← Base model with common fields
│   │   ├── customer.py          ← Customer entity and related models
│   │   ├── plan.py              ← Plan and pricing models
│   │   ├── subscription.py      ← Subscription lifecycle models
│   │   ├── invoice.py           ← Invoice and line item models
│   │   ├── payment.py           ← Payment tracking models
│   │   └── usage.py             ← Usage tracking models
│   ├── types/
│   │   ├── __init__.py          ← Money type with currency handling
│   │   ├── money.py             ← All system enums
│   │   ├── enums.py             ← Audit event types and logging
│   ├── database/
│   │   ├── __init__.py          ← Database connection management
│   │   └── migrations/          ← Alembic migration files
│   └── config/
│       ├── __init__.py          ← Database and system configuration
│       └── settings.py
└── tests/
    ├── test_models/            ← Unit tests for each entity
    └── test_integration/       ← Database integration tests
└── requirements.txt
```

Core Types Implementation

Complete Money Type Implementation:

```
# src/types/money.py
```

PYTHON

```
from decimal import Decimal, ROUND_HALF_UP

from typing import Dict, Optional

import json

# Currency configuration with precision and display information

SUPPORTED_CURRENCIES: Dict[str, Dict[str, any]] = {

    'USD': {'name': 'US Dollar', 'symbol': '$', 'decimal_places': 2},

    'EUR': {'name': 'Euro', 'symbol': '€', 'decimal_places': 2},

    'GBP': {'name': 'British Pound', 'symbol': '£', 'decimal_places': 2},

    'JPY': {'name': 'Japanese Yen', 'symbol': '¥', 'decimal_places': 0},

}

DEFAULT_CURRENCY = 'USD'

DECIMAL_PRECISION = Decimal('0.01')


class Money:

    """
    Immutable money type that stores amounts in smallest currency unit (cents)
    to avoid floating-point precision errors in financial calculations.
    """

    def __init__(self, amount_cents: int, currency_code: str):

        if currency_code not in SUPPORTED_CURRENCIES:

            raise ValueError(f"Unsupported currency: {currency_code}")

        self.amount_cents = amount_cents

        self.currency_code = currency_code.upper()
```

```
@classmethod

def from_decimal(cls, amount: Decimal, currency: str) -> 'Money':

    """
    Create Money instance from decimal amount (e.g., 19.99 USD).

    Rounds to appropriate precision for the currency.

    """

    currency_info = SUPPORTED_CURRENCIES[currency.upper()]

    decimal_places = currency_info['decimal_places']

    # Convert to smallest unit (e.g., cents for USD)

    multiplier = Decimal(10 ** decimal_places)

    amount_cents = int((amount * multiplier).quantize(Decimal('1'),
rounding=ROUND_HALF_UP))

    return cls(amount_cents, currency)


def to_decimal(self) -> Decimal:

    """
    Convert to decimal representation (e.g., 1999 cents -> 19.99).

    """

    currency_info = SUPPORTED_CURRENCIES[self.currency_code]

    decimal_places = currency_info['decimal_places']

    divisor = Decimal(10 ** decimal_places)

    return Decimal(self.amount_cents) / divisor
```

```
def __add__(self, other: 'Money') -> 'Money':  
  
    if self.currency_code != other.currency_code:  
  
        raise ValueError("Cannot add different currencies")  
  
    return Money(self.amount_cents + other.amount_cents, self.currency_code)  
  
  
def __sub__(self, other: 'Money') -> 'Money':  
  
    if self.currency_code != other.currency_code:  
  
        raise ValueError("Cannot subtract different currencies")  
  
    return Money(self.amount_cents - other.amount_cents, self.currency_code)  
  
  
def __eq__(self, other: 'Money') -> bool:  
  
    return (self.amount_cents == other.amount_cents and  
  
            self.currency_code == other.currency_code)  
  
  
def __repr__(self) -> str:  
  
    return f"Money({self.amount_cents}, '{self.currency_code}')"  
  
  
def __str__(self) -> str:  
  
    currency_info = SUPPORTED_CURRENCIES[self.currency_code]  
  
    symbol = currency_info['symbol']  
  
    amount = self.to_decimal()  
  
    return f"{symbol}{amount}"
```

Enum Definitions:

```
# src/types/enums.py
```

PYTHON

```
from enum import Enum


class SubscriptionStatus(Enum):

    TRIAL = "trial"

    ACTIVE = "active"

    PAST_DUE = "past_due"

    CANCELLED = "cancelled"

    EXPIRED = "expired"


class PlanStatus(Enum):

    ACTIVE = "active"

    DEPRECATED = "deprecated"

    ARCHIVED = "archived"


class PricingModel(Enum):

    FLAT_RATE = "flat_rate"

    PER_SEAT = "per_seat"

    TIERED = "tiered"

    VOLUME = "volume"


class BillingInterval(Enum):

    MONTHLY = "monthly"

    QUARTERLY = "quarterly"

    ANNUAL = "annual"


class InvoiceStatus(Enum):

    DRAFT = "draft"

    OPEN = "open"
```

```
PAID = "paid"

VOID = "void"

UNCOLLECTIBLE = "uncollectible"

class PaymentStatus(Enum):

    PENDING = "pending"

    SUCCEEDED = "succeeded"

    FAILED = "failed"

    CANCELLED = "cancelled"

    REFUNDED = "refunded"

class AuditEventType(Enum):

    SUBSCRIPTION_CREATED = "subscription_created"

    SUBSCRIPTION_CANCELLED = "subscription_cancelled"

    PLAN_CHANGED = "plan_changed"

    INVOICE_GENERATED = "invoice_generated"

    PAYMENT_PROCESSED = "payment_processed"

    USAGE_RECORDED = "usage_recorded"
```

Database Connection Management:

```
# src/database/connection.py
```

PYTHON

```
from sqlalchemy import create_engine, event
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker, Session
from contextlib import contextmanager
from typing import Generator
import os

# Database isolation levels for financial operations
ISOLATION_LEVEL_SERIALIZABLE = "SERIALIZABLE"

class DatabaseManager:

    def __init__(self, connection_string: str):
        self.connection_string = connection_string
        self.engine = create_engine(
            connection_string,
            echo=os.getenv("DEBUG_SQL", "false").lower() == "true",
            pool_pre_ping=True, # Validate connections before use
            pool_recycle=3600, # Refresh connections hourly
        )
        self.SessionLocal = sessionmaker(bind=self.engine)

    @contextmanager
    def transaction(self, isolation_level: str = ISOLATION_LEVEL_SERIALIZABLE) -> Generator[Session, None, None]:
        """
        Provide database transaction with specified isolation level.

        Financial operations should use SERIALIZABLE to prevent race conditions.
        
```

```
""""

session = self.SessionLocal()

session.connection(execution_options={"isolation_level": isolation_level})
```



```
try:

    yield session

    session.commit()

except Exception:

    session.rollback()

    raise

finally:

    session.close()

# Base model with common fields

Base = declarative_base()

# Global database manager instance

db_manager = DatabaseManager(os.getenv("DATABASE_URL", "postgresql://localhost/billing"))
```

Base Model with Audit Trail:

```
# src/models/base.py
```

PYTHON

```
from sqlalchemy import Column, DateTime, String, Text
from sqlalchemy.dialects.postgresql import UUID, JSONB
from sqlalchemy.sql import func
from database.connection import Base
from types.audit import AuditEventType
import uuid
from datetime import datetime
from typing import Dict, Any, Optional

class BaseModel(Base):
    __abstract__ = True

    # Common fields for all entities
    created_at = Column(DateTime(timezone=True), server_default=func.now(), nullable=False)
    updated_at = Column(DateTime(timezone=True), server_default=func.now(),
onupdate=func.now(), nullable=False)
    metadata = Column(JSONB, default=dict, nullable=False)

    def log_event(
        event_type: AuditEventType,
        entity_id: str,
        entity_type: str,
        changes: Dict[str, Any],
        actor_id: Optional[str] = None,
        metadata: Optional[Dict[str, Any]] = None
    ) -> None:
        """

```

```
Record audit trail event for billing operations.

This is a placeholder - implement with your audit logging system.

"""

audit_record = {

    "event_type": event_type.value,

    "entity_id": entity_id,

    "entity_type": entity_type,

    "changes": changes,

    "actor_id": actor_id,

    "metadata": metadata or {},

    "timestamp": datetime.utcnow().isoformat(),

}

# TODO: Implement actual audit logging

# Options: database table, external audit service, structured logs

print(f"AUDIT: {audit_record}")
```

Core Entity Skeletons

Customer Model:

```
# src/models/customer.py
```

PYTHON

```
from sqlalchemy import Column, String, Integer, Boolean
from sqlalchemy.dialects.postgresql import UUID, JSONB
from models.base import BaseModel
from types.enums import AccountStatus
import uuid

class Customer(BaseModel):
    __tablename__ = 'customers'

    customer_id = Column(UUID(as_uuid=True), primary_key=True, default=uuid.uuid4)
    external_id = Column(String(255), unique=True, nullable=True)
    email = Column(String(255), nullable=False, index=True)
    name = Column(String(255), nullable=False)

    # TODO: Implement billing address as structured JSON
    billing_address = Column(JSONB, nullable=True)

    # TODO: Add tax_id field with validation
    tax_id = Column(String(50), nullable=True)

    currency = Column(String(3), nullable=False, default='USD')
    credit_balance_cents = Column(Integer, nullable=False, default=0)

    # TODO: Integrate with payment gateway for payment method storage
    payment_method_id = Column(String(255), nullable=True)
```

```
billing_email = Column(String(255), nullable=True)

account_status = Column(String(20), nullable=False, default='active')

def apply_credit(self, amount_cents: int) -> None:

    """
    Add credit to customer balance. Used for refunds and downgrades.

    """

    # TODO: Validate amount is positive

    # TODO: Log credit application event

    # TODO: Update credit_balance_cents

    pass


def consume_credit(self, amount_cents: int) -> int:

    """
    Apply available credit to reduce invoice amount.

    Returns amount of credit actually applied.

    """

    # TODO: Calculate available credit to apply (min of requested and available)

    # TODO: Update credit balance

    # TODO: Log credit consumption event

    # TODO: Return amount applied

    pass
```

Plan Model:

```
# src/models/plan.py
```

PYTHON

```
from sqlalchemy import Column, String, Integer, Text, Boolean

from sqlalchemy.dialects.postgresql import UUID, JSONB

from models.base import BaseModel

from types.enums import PlanStatus, BillingInterval, PricingModel

import uuid


class Plan(BaseModel):

    __tablename__ = 'plans'

    plan_id = Column(UUID(as_uuid=True), primary_key=True, default=uuid.uuid4)

    plan_code = Column(String(100), nullable=False, unique=True)

    name = Column(String(255), nullable=False)

    description = Column(Text, nullable=True)

    status = Column(String(20), nullable=False, default=PlanStatus.ACTIVE.value)

    billing_interval = Column(String(20), nullable=False)

    billing_interval_count = Column(Integer, nullable=False, default=1)

    pricing_model = Column(String(20), nullable=False)

    base_price_cents = Column(Integer, nullable=False)

    currency = Column(String(3), nullable=False, default='USD')

    trial_period_days = Column(Integer, nullable=True)

    setup_fee_cents = Column(Integer, nullable=False, default=0)
```

```
# Structured data for features and pricing

features = Column(JSONB, default=list, nullable=False)

usage_limits = Column(JSONB, default=dict, nullable=False)

pricing_tiers = Column(JSONB, default=list, nullable=False)

def calculate_usage_charge(self, usage_type: str, quantity: float) -> int:

    """
    Calculate charges for usage beyond plan limits.

    Returns charge amount in cents.

    """

    # TODO: Get usage limit for this usage_type from usage_limits

    # TODO: Calculate overage quantity (usage - limit)

    # TODO: Apply tiered pricing from pricing_tiers

    # TODO: Return total charge in cents

    pass


def has_feature(self, feature_name: str) -> bool:

    """
    Check if plan includes specified feature.

    """

    # TODO: Check if feature_name exists in features list

    pass


def is_available_for_signup(self) -> bool:

    """
    Check if plan can be used for new subscriptions.

    """


```

```
# TODO: Return True only if status is ACTIVE  
pass
```

Milestone Checkpoints

Checkpoint 1: Basic Entity Creation After implementing the core entities, verify the data model works correctly:

```
# Run basic model tests  
  
python -m pytest tests/test_models/ -v  
  
# Test money type operations  
  
python -c "  
  
from src.types.money import Money  
  
from decimal import Decimal  
  
# Create money instances  
  
usd_19_99 = Money.from_decimal(Decimal('19.99'), 'USD')  
  
usd_5_00 = Money.from_decimal(Decimal('5.00'), 'USD')  
  
# Test arithmetic  
  
total = usd_19_99 + usd_5_00  
  
print(f'Total: {total}') # Should print: $24.99  
  
# Test precision  
  
cents_1999 = Money(1999, 'USD')  
  
print(f'From cents: {cents_1999}') # Should print: $19.99  
"  
"
```

BASH

Expected output shows proper money arithmetic without precision errors.

Checkpoint 2: Database Schema Creation Verify the database schema generates correctly:

```
# Generate migration for core entities
alembic revision --autogenerate -m "Add core billing entities"

# Apply migration
alembic upgrade head

# Verify tables were created
psql $DATABASE_URL -c "\dt"
```

BASH

You should see tables for customers, plans, subscriptions, invoices, invoice_line_items, payments, usage_events, usage_aggregations, and usage_billing.

Checkpoint 3: Entity Relationship Validation Test that foreign key relationships work correctly:

```
# Test script to verify relationships

from src.models import Customer, Plan, Subscription

from src.database.connection import db_manager

from src.types.money import Money

from decimal import Decimal


with db_manager.transaction() as session:

    # Create test customer

    customer = Customer(
        email="test@example.com",
        name="Test Customer",
        currency="USD"
    )

    session.add(customer)


    # Create test plan

    plan = Plan(
        plan_code="test-monthly",
        name="Test Monthly Plan",
        billing_interval="monthly",
        pricing_model="flat_rate",
        base_price_cents=1999,  # $19.99
        currency="USD"
    )

    session.add(plan)

    session.flush() # Get IDs before creating subscription
```

```
# Create subscription linking customer and plan

subscription = Subscription(
    customer_id=customer.customer_id,
    plan_id=plan.plan_id,
    status="active"
)

session.add(subscription)

print("✓ Successfully created related entities")
```

This checkpoint confirms that the core data model supports the relationships needed for subscription billing workflows.

Plan Management Component

Milestone(s): Milestone 1 (Plans & Pricing) - implements flexible pricing plans with tiers, features, and currencies

Mental Model: Product Catalog Design

Think of plan management like designing a product catalog for a large retail chain. Just as a store needs a master catalog that defines each product's price, features, and availability across different locations, a subscription system needs a plan catalog that defines pricing models, feature entitlements, and billing terms across different customer segments.

Consider how Apple manages its iPhone product line. They maintain multiple models (iPhone 15, iPhone 15 Pro, iPhone 15 Pro Max) with different storage tiers (128GB, 256GB, 512GB, 1TB). Each combination has a specific price point and set of features. When Apple releases the iPhone 16, they don't delete the iPhone 15 from existence - existing customers keep their devices and support contracts. Similarly, when you deprecate a subscription plan, existing subscribers must be "grandfathered" into their original terms.

The catalog also defines what features each product tier unlocks. An iPhone 15 Pro includes features that the base iPhone 15 doesn't have (ProRAW photography, macro lens, etc.). In subscription systems, this translates to feature entitlements - a Premium plan might include API access limits of 10,000 requests per month, while a Basic plan only allows 1,000 requests.

Just as retailers need to handle currency conversion for international customers (a \$999 iPhone costs €1,129 in Europe due to taxes and exchange rates), subscription plans must support multiple currencies with appropriate regional pricing. The key insight is that you're not just storing prices - you're building a comprehensive product management system that governs what customers can access and how much they pay.

Pricing Model Implementation

The plan management system must support three fundamental pricing models, each serving different business use cases and customer preferences. Understanding these models is crucial because each requires different calculation logic, validation rules, and upgrade/downgrade behavior.

Flat-Rate Pricing represents the simplest subscription model where customers pay a fixed amount regardless of usage. Think of Netflix's monthly subscription - whether you watch one movie or fifty, you pay the same \$15.99. This model provides predictable revenue for the business and predictable costs for customers. The implementation stores a single `base_price_cents` value per plan, and billing calculations are straightforward multiplication based on the billing interval.

Tiered Pricing creates multiple usage brackets with different rates per bracket, similar to progressive tax systems. Consider a cloud storage service where the first 100GB costs \$5, the next 400GB costs \$0.03 per GB, and anything beyond 500GB costs \$0.02 per GB. A customer using 750GB would pay: \$5 (first tier) + \$12 ($400\text{GB} \times \$0.03$) + \$5 ($250\text{GB} \times \$0.02$) = \$22 total. Each usage unit is charged at the rate of its respective tier.

Volume Pricing applies a single rate to the entire usage amount based on which tier the total usage falls into. Using the same cloud storage example, if a customer uses 750GB, they would pay the entire 750GB at the \$0.02 rate (since they're in the highest tier), resulting in a \$15 charge. This model rewards higher usage with better rates across all consumption.

The distinction between tiered and volume pricing is critical and frequently misunderstood. Tiered pricing charges each bracket separately (like tax brackets), while volume pricing applies one rate to the entire amount (like bulk discounts). The Plan entity must clearly specify which model applies to avoid billing errors.

Pricing Model	Description	Calculation Method	Use Case Example
Flat-Rate	Fixed charge per billing period	<code>base_price_cents × billing_periods</code>	Netflix subscription
Tiered	Different rates per usage bracket	Sum of $(\text{bracket_usage} \times \text{bracket_rate})$	Progressive API pricing
Volume	Single rate based on total usage tier	<code>total_usage × applicable_tier_rate</code>	Bulk storage discounts

Feature Entitlements represent what capabilities each plan unlocks for subscribers. Unlike pricing, which determines charges, entitlements determine access control throughout your application. A plan defines both what a customer pays and what they can do.

Feature entitlements should be designed as a flexible key-value system where each plan maps to a set of feature flags and their corresponding limits. For example, a "Starter" plan might include

```
api_requests_per_month: 1000, max_team_members: 5, advanced_analytics: false, while a  
"Professional" plan might include api_requests_per_month: 10000, max_team_members: 25,  
advanced_analytics: true.
```

The Plan entity structure must accommodate all these pricing models while maintaining clarity about which model applies:

Field	Type	Description
plan_id	UUID	Unique identifier for the plan
plan_code	str	Human-readable code (e.g., "starter-monthly")
name	str	Display name (e.g., "Starter Plan")
description	str	Marketing description of the plan
billing_interval	str	Frequency of charges (monthly, yearly, quarterly)
pricing_model	str	Type of pricing (flat_rate, tiered, volume)
base_price_cents	int	Base price in smallest currency unit
currency	str	Currency code (USD, EUR, GBP)
trial_period_days	int	Number of free trial days (0 if no trial)
feature_entitlements	JSON	Key-value mapping of features to limits
pricing_tiers	JSON	Array of tier definitions for tiered/volume pricing
is_active	bool	Whether new subscriptions can use this plan
created_at	timestamp	Plan creation time
deprecated_at	timestamp	When plan was deprecated (null if active)

The `pricing_tiers` field for tiered and volume pricing contains an array of tier objects:

Tier Field	Type	Description
<code>tier_number</code>	int	Order of the tier (1, 2, 3, etc.)
<code>min_quantity</code>	int	Minimum usage for this tier
<code>max_quantity</code>	int	Maximum usage for this tier (null for unlimited)
<code>price_per_unit_cents</code>	int	Cost per unit in this tier
<code>flat_fee_cents</code>	int	Fixed fee for entering this tier

ADR: Plan Versioning Strategy

Decision: Immutable Plan Versioning with Grandfathering

- **Context:** When businesses need to change plan pricing or features, they face a dilemma. Updating existing plans affects current subscribers, potentially violating their original agreement or creating customer dissatisfaction. However, maintaining multiple plan versions creates complexity in the system and administrative overhead.
- **Options Considered:**
 1. Mutable plans with forced migrations
 2. Plan versioning with optional migrations
 3. Immutable plans with grandfathering
- **Decision:** Implement immutable plan versioning where plan changes create new versions, and existing subscribers remain on their original plan version unless they explicitly choose to migrate.
- **Rationale:** This approach honors existing customer agreements while allowing business flexibility to adjust pricing and features for new customers. It prevents unexpected billing changes that could damage customer relationships and ensures legal compliance with subscription terms.
- **Consequences:** Increases system complexity as multiple plan versions must be maintained. Customer support must understand which customers are on which plan versions. Reporting and analytics become more complex when aggregating across plan versions.

Option	Pros	Cons
Mutable Plans	Simple data model, single plan per type	Breaks existing customer agreements, potential legal issues
Optional Migrations	Balanced approach, customer choice	Complex migration logic, unclear customer experience
Immutable Versioning	Honors agreements, clear audit trail	Multiple versions to maintain, complex reporting

The immutable versioning strategy requires extending the Plan entity to support version tracking. When a plan needs modification, the system creates a new plan record with an incremented version number, marking the previous version as deprecated for new subscriptions while keeping it available for existing subscribers.

Plan Version Management introduces additional fields to track the relationship between plan versions:

Field	Type	Description
<code>plan_family_id</code>	UUID	Groups all versions of the same logical plan
<code>version_number</code>	int	Sequential version within the plan family
<code>supersedes_plan_id</code>	UUID	Previous version this plan replaces
<code>migration_eligible</code>	bool	Whether existing subscribers can upgrade to this version
<code>migration_incentive</code>	JSON	Special offers for customers who migrate

The plan family concept allows grouping related plan versions while maintaining clear lineage. For example, all versions of the "Professional Plan" share the same `plan_family_id` but have different `version_number` values. This enables reporting across all Professional subscribers while maintaining version-specific billing logic.

Grandfathering Rules define how existing subscribers interact with plan changes:

- Price Protection:** Existing subscribers maintain their original pricing unless they voluntarily change plans
- Feature Evolution:** New features may be added to all versions of a plan family, but existing features cannot be removed
- Billing Interval Lock:** Subscribers cannot be forced to change from monthly to yearly billing or vice versa
- Migration Windows:** When offering upgrades, provide limited-time migration opportunities with clear terms

Common Pitfalls in Plan Management

⚠ Pitfall: Currency Float Arithmetic Using floating-point arithmetic for currency calculations introduces rounding errors that accumulate over many transactions. A plan priced at \$9.99 might be stored as 9.989999...

internally, leading to billing discrepancies. Always store prices as integers in the smallest currency unit (cents for USD, pence for GBP). Use the `Money` type consistently throughout the system and convert to decimal representation only for display purposes.

⚠ Pitfall: Plan Deletion Instead of Deprecation Deleting old plans breaks the referential integrity of existing subscriptions and makes historical reporting impossible. Instead of deletion, mark plans as deprecated with a timestamp. Deprecated plans should not be available for new subscriptions but must remain queryable for existing subscribers. This maintains data consistency and supports customer service inquiries about historical billing.

⚠ Pitfall: Inconsistent Feature Entitlement Checking Checking subscription status (`active`) instead of plan features (`has_feature`) leads to incorrect access control. An active subscription to a Basic plan should not grant access to Premium features, even if the subscription is current on payments. Always check the specific feature entitlement rather than subscription status alone. This prevents feature access bugs and ensures proper plan enforcement.

⚠ Pitfall: Timezone-Naive Billing Intervals Storing billing intervals as simple strings like "monthly" without considering timezone effects creates ambiguity about when billing occurs. A customer in Tokyo subscribing on January 31st might have their billing date shift unexpectedly due to timezone calculations. Store billing anchor dates as UTC timestamps and perform timezone conversion only for display purposes.

⚠ Pitfall: Hard-Coded Feature Lists Embedding feature names directly in application code makes plan management inflexible and requires code deployments to add new features. Instead, use a dynamic feature entitlement system where plans reference configurable feature keys. This allows business users to create new plans and modify feature access without engineering involvement.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
Plan Storage	SQLite with JSON columns	PostgreSQL with JSONB indexing
Feature Flags	Simple key-value in plan JSON	Redis with feature flag service
Currency Handling	Python Decimal with hardcoded rates	Integration with currency exchange API
Plan Validation	Pydantic models with basic validation	Marshmallow with custom business rules
Caching	In-memory Python dict	Redis with TTL-based invalidation

Recommended File Structure

```
subscription_system/
  core/
    models/
      plan.py           ← Plan entity and validation
      money.py          ← Money type and currency handling
    services/
      plan_service.py   ← Plan management business logic
      feature_service.py ← Feature entitlement checking
    repositories/
      plan_repository.py ← Plan data access layer
  api/
    plan_api.py        ← REST endpoints for plan management
  migrations/
    001_create_plans.sql ← Database schema creation
  tests/
    test_plan_models.py  ← Unit tests for plan logic
    test_plan_service.py ← Service layer tests
```

Infrastructure Starter Code

```
# core/models/money.py
```

PYTHON

```
from decimal import Decimal, ROUND_HALF_UP

from typing import Dict


SUPPORTED_CURRENCIES: Dict[str, Dict[str, str]] = {

    'USD': {'symbol': '$', 'name': 'US Dollar', 'decimal_places': 2},

    'EUR': {'symbol': '€', 'name': 'Euro', 'decimal_places': 2},

    'GBP': {'symbol': '£', 'name': 'British Pound', 'decimal_places': 2},

    'JPY': {'symbol': '¥', 'name': 'Japanese Yen', 'decimal_places': 0},

}

DEFAULT_CURRENCY = 'USD'

DECIMAL_PRECISION = Decimal('0.01')


class Money:

    """Represents a monetary amount with currency, stored as integer cents."""

    def __init__(self, amount_cents: int, currency_code: str):

        if currency_code not in SUPPORTED_CURRENCIES:

            raise ValueError(f"Unsupported currency: {currency_code}")

        self.amount_cents = amount_cents

        self.currency_code = currency_code

    @classmethod

    def from_decimal(cls, amount: Decimal, currency: str) -> 'Money':

        """Create Money from decimal amount, converting to cents."""

        if currency not in SUPPORTED_CURRENCIES:

            raise ValueError(f"Unsupported currency: {currency}")
```

```
    decimal_places = SUPPORTED_CURRENCIES[currency]['decimal_places']

    multiplier = Decimal(10 ** decimal_places)

    amount_cents = int(amount.quantize(DECIMAL_PRECISION, rounding=ROUND_HALF_UP) *
multiplier)

    return cls(amount_cents, currency)

def to_decimal(self) -> Decimal:

    """Convert Money to decimal representation for display."""

    decimal_places = SUPPORTED_CURRENCIES[self.currency_code]['decimal_places']

    divisor = Decimal(10 ** decimal_places)

    return Decimal(self.amount_cents) / divisor

def __add__(self, other: 'Money') -> 'Money':

    if self.currency_code != other.currency_code:

        raise ValueError("Cannot add different currencies")

    return Money(self.amount_cents + other.amount_cents, self.currency_code)

def __str__(self) -> str:

    symbol = SUPPORTED_CURRENCIES[self.currency_code]['symbol']

    return f"{symbol}{self.to_decimal()}"
```

core/models/plan.py

```
from uuid import UUID, uuid4

from datetime import datetime

from typing import Dict, List, Optional, Any

from enum import Enum

from dataclasses import dataclass, field
```

```
class PricingModel(Enum):

    FLAT_RATE = "flat_rate"

    TIERED = "tiered"

    VOLUME = "volume"


class BillingInterval(Enum):

    MONTHLY = "monthly"

    YEARLY = "yearly"

    QUARTERLY = "quarterly"


@dataclass

class PricingTier:

    """Represents a single tier in tiered or volume pricing."""

    tier_number: int

    min_quantity: int

    max_quantity: Optional[int] # None for unlimited

    price_per_unit_cents: int

    flat_fee_cents: int = 0


@dataclass

class Plan:

    """Core plan entity with pricing and feature information."""

    plan_id: UUID = field(default_factory=uuid4)

    plan_code: str = ""

    name: str = ""

    description: str = ""

    billing_interval: str = BillingInterval.MONTHLY.value

    pricing_model: str = PricingModel.FLAT_RATE.value
```

```
base_price_cents: int = 0

currency: str = DEFAULT_CURRENCY

trial_period_days: int = 0

feature_entitlements: Dict[str, Any] = field(default_factory=dict)

pricing_tiers: List[PricingTier] = field(default_factory=list)

is_active: bool = True

plan_family_id: Optional[UUID] = None

version_number: int = 1

supersedes_plan_id: Optional[UUID] = None

migration_eligible: bool = True

created_at: datetime = field(default_factory=datetime.utcnow)

deprecated_at: Optional[datetime] = None


def has_feature(self, feature_name: str) -> bool:

    """Check if this plan includes the specified feature."""

    # TODO: Implement feature checking logic

    # TODO: Handle boolean features (return feature_entitlements.get(feature_name,
    False))

    # TODO: Handle numeric limits (return feature_entitlements.get(feature_name, 0) >
    0)

    # TODO: Consider feature inheritance from plan family

    pass


def get_feature_limit(self, feature_name: str) -> Optional[int]:

    """Get the numeric limit for a feature, or None if unlimited."""

    # TODO: Extract numeric limits from feature_entitlements

    # TODO: Handle special case of -1 or None meaning unlimited

    # TODO: Return 0 if feature is not included in this plan
```

```
pass

def calculate_base_charge(self) -> Money:
    """Calculate the base recurring charge for this plan."""
    # TODO: Return Money object with base_price_cents and currency
    # TODO: Handle case where pricing_model is not flat_rate (should return 0)
    pass

# Database repository implementation

class PlanRepository:
    """Data access layer for plan operations."""

    def __init__(self, db_manager):
        self.db = db_manager

    def create_plan(self, plan: Plan) -> Plan:
        """Create a new plan in the database."""
        # TODO: Validate plan data before insertion
        # TODO: Check for unique plan_code within plan_family
        # TODO: Insert plan record with all fields
        # TODO: Handle database constraints and return created plan
        pass

    def get_plan_by_id(self, plan_id: UUID) -> Optional[Plan]:
        """Retrieve a plan by its ID."""
        # TODO: Query database for plan with given ID
        # TODO: Convert database row to Plan object
```

```
# TODO: Handle case where plan is not found
pass

def get_active_plans(self) -> List[Plan]:
    """Get all plans available for new subscriptions."""

    # TODO: Query for plans where is_active = True and deprecated_at IS NULL

    # TODO: Convert database rows to Plan objects

    # TODO: Order by plan_family_id and version_number

    pass

def deprecate_plan(self, plan_id: UUID) -> bool:
    """Mark a plan as deprecated, preventing new subscriptions."""

    # TODO: Set deprecated_at timestamp to current time

    # TODO: Set is_active to False

    # TODO: Ensure existing subscriptions are not affected

    # TODO: Return True if plan was found and updated

    pass
```

Core Logic Skeleton

```
# core/services/plan_service.py
```

PYTHON

```
from typing import List, Optional

from uuid import UUID

from core.models.plan import Plan, PricingModel, Money

from core.repositories.plan_repository import PlanRepository


class PlanService:

    """Business logic for plan management operations."""

    def __init__(self, plan_repo: PlanRepository):
        self.plan_repo = plan_repo

    def create_plan_family(self, plan_data: dict) -> Plan:
        """Create the first version of a new plan family."""

        # TODO: Validate required fields (name, plan_code, pricing_model, etc.)

        # TODO: Generate new plan_family_id (same as plan_id for first version)

        # TODO: Set version_number to 1

        # TODO: Validate pricing_model and ensure required pricing data exists

        # TODO: Create Plan object and save via repository

        # TODO: Return created plan

        pass

    def create_plan_version(self, plan_family_id: UUID, updated_data: dict) -> Plan:
        """Create a new version of an existing plan family."""

        # TODO: Get latest version of the plan family

        # TODO: Validate that changes warrant a new version

        # TODO: Increment version_number
```

```
# TODO: Set supersedes_plan_id to previous version

# TODO: Deprecate previous version if specified

# TODO: Create and save new plan version

pass


def calculate_plan_charge(self, plan: Plan, usage_quantity: int = 0) -> Money:

    """Calculate total charge for a plan given usage quantity."""

    # TODO: Handle flat_rate pricing (return base_price_cents)

    # TODO: Handle tiered pricing (sum charges across tiers)

    # TODO: Handle volume pricing (find tier and apply rate to total)

    # TODO: Add any flat fees from pricing tiers

    # TODO: Return Money object with calculated amount

    pass


def validate_plan_upgrade(self, from_plan_id: UUID, to_plan_id: UUID) -> bool:

    """Validate that a plan change is allowed."""

    # TODO: Check that both plans exist and are in same family OR different families

    # TODO: Verify to_plan allows migrations (migration_eligible = True)

    # TODO: Ensure currency compatibility

    # TODO: Check business rules (e.g., no downgrades, version restrictions)

    # TODO: Return True if upgrade is valid

    pass


# core/services/feature_service.py

class FeatureService:

    """Service for checking feature entitlements and usage limits."""
```

```
def __init__(self, plan_repo: PlanRepository):

    self.plan_repo = plan_repo


def check_feature_access(self, plan_id: UUID, feature_name: str) -> bool:

    """Check if a plan grants access to a specific feature."""

    # TODO: Retrieve plan from repository

    # TODO: Call plan.has_feature(feature_name)

    # TODO: Handle case where plan is not found (return False)

    # TODO: Log access check for auditing purposes

    pass


def get_feature_usage_limit(self, plan_id: UUID, feature_name: str) -> Optional[int]:

    """Get the usage limit for a feature, or None if unlimited."""

    # TODO: Retrieve plan from repository

    # TODO: Call plan.get_feature_limit(feature_name)

    # TODO: Handle inheritance from plan family if configured

    # TODO: Return limit or None for unlimited access

    pass


def validate_feature_usage(self, plan_id: UUID, feature_name: str, proposed_usage: int)
-> bool:

    """Check if proposed usage is within plan limits."""

    # TODO: Get feature limit for the plan

    # TODO: Handle unlimited access (limit is None)

    # TODO: Compare proposed_usage against limit

    # TODO: Return True if usage is within limits

    pass
```

Milestone Checkpoint

After implementing the plan management component, verify the following functionality:

Test Commands:

```
# Run unit tests for plan models and services                                BASH
python -m pytest tests/test_plan_models.py -v
python -m pytest tests/test_plan_service.py -v

# Test plan creation via API
curl -X POST http://localhost:8000/api/plans \
-H "Content-Type: application/json" \
-d '{
  "plan_code": "starter-monthly",
  "name": "Starter Plan",
  "billing_interval": "monthly",
  "base_price_cents": 999,
  "currency": "USD",
  "feature_entitlements": {
    "api_requests_per_month": 1000,
    "max_team_members": 5
  }
}'
```

Expected Behavior:

1. Plan creation should return a complete Plan object with generated `plan_id` and `plan_family_id`
2. Feature checking should correctly identify which features are included in each plan
3. Pricing calculations should handle different pricing models accurately
4. Plan versioning should create new versions without affecting existing plans
5. Currency handling should maintain precision without floating-point errors

Signs of Issues:

- **Plan creation fails:** Check required field validation and database constraints
- **Currency precision errors:** Verify all prices are stored as integers in cents
- **Feature access bugs:** Ensure feature checking uses plan entitlements, not subscription status
- **Version conflicts:** Check that plan_family_id relationships are maintained correctly

Subscription Lifecycle Management

Milestone(s): Milestone 2 (Subscription Lifecycle) - implements subscription creation, activation, renewal, and cancellation with appropriate state transitions and billing cycle management

Mental Model: Membership Lifecycle

Think of subscription management like running a premium fitness club with sophisticated membership tiers. When someone joins your gym, they don't just pay once and disappear - they enter into an ongoing relationship with predictable patterns and lifecycle events.

Consider how gym memberships work in the real world. A new member might start with a free trial week to test the facilities. During this trial, they have access to basic equipment but can't use premium services like personal training or spa facilities. If they don't cancel before the trial ends, their membership automatically converts to a paid plan - perhaps a monthly membership at \$50 per month.

Once they're an active paying member, several things can happen. They might upgrade to a premium plan to access the spa and personal training. They could downgrade to a basic plan if money gets tight. They might go on vacation and pause their membership for a month. Eventually, they might cancel - either immediately (forfeiting any prepaid time) or at the end of their current billing period.

Throughout this lifecycle, the gym must track payment status carefully. If a member's credit card fails, they don't immediately lose access - there's a grace period where the gym tries to collect payment while the member can still use facilities. Only after multiple failed attempts does the membership move to a suspended state.

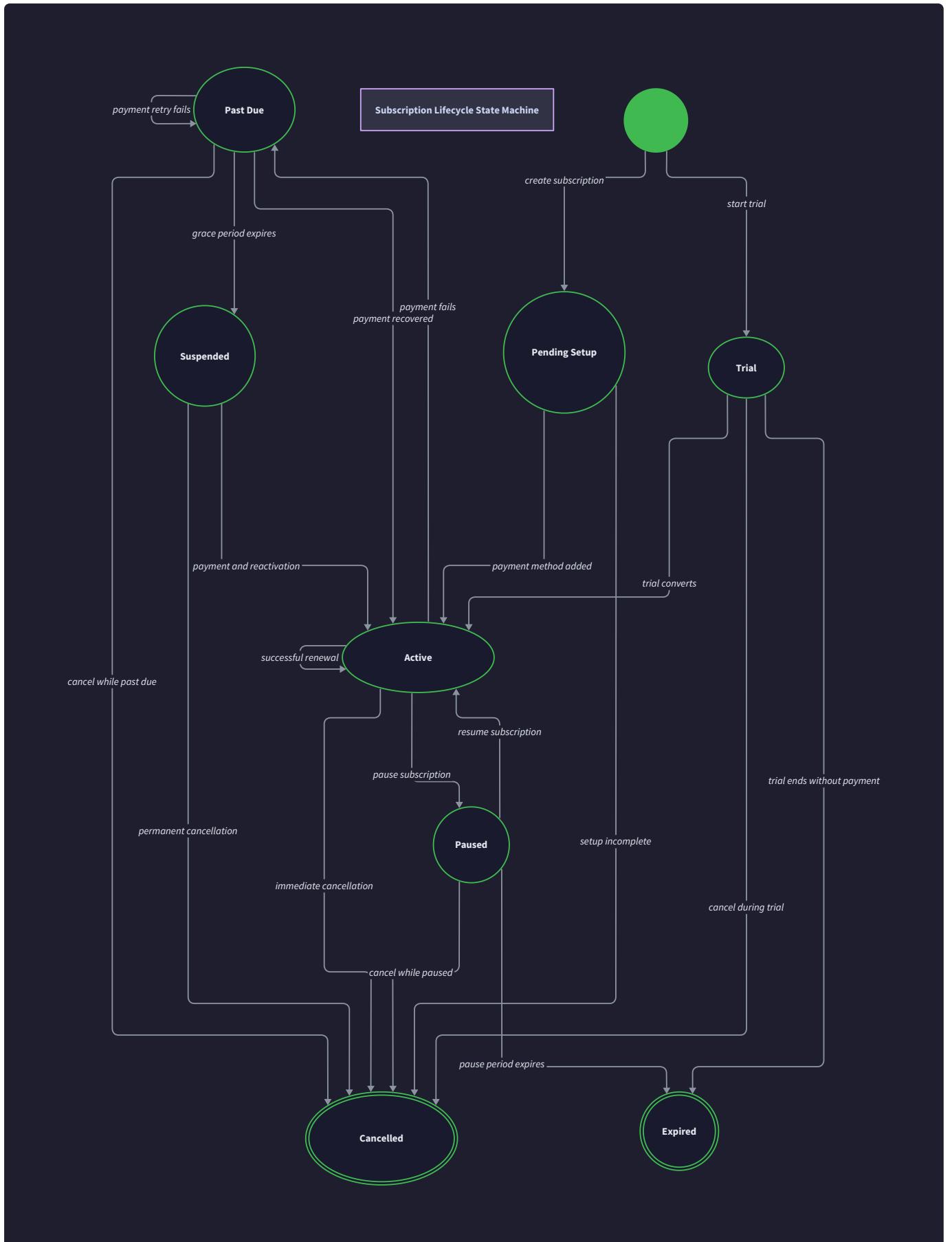
This is exactly how subscription billing works. Each **subscription** represents an ongoing relationship between a customer and a service, with predictable state transitions, billing cycles, and lifecycle events. The subscription moves through various states (trial, active, past_due, cancelled) based on payment success, customer actions, and business rules.

The key insight is that subscriptions are not just payment records - they're stateful entities that evolve over time according to business logic. Managing this lifecycle correctly ensures customers have a smooth experience while protecting business revenue through proper dunning management and retention policies.

Subscription State Machine

A subscription's journey follows a well-defined state machine where each state represents a distinct phase of the customer relationship. Understanding these states and their transitions is crucial for implementing reliable billing logic and providing clear customer communication.

The subscription lifecycle begins when a customer selects a plan and provides payment information. Unlike simple e-commerce transactions, subscriptions must track ongoing state because the relationship extends far beyond the initial purchase. Each state change triggers specific business logic, from provisioning access to sending dunning emails.



Here are the core subscription states and their business meaning:

State	Description	Customer Access	Billing Status	Duration
<code>incomplete</code>	Subscription created but payment method not yet confirmed	No access	Payment pending	Until payment succeeds/fails
<code>trialing</code>	Free trial period active	Limited access per plan	No charges	Until trial_end_date
<code>active</code>	Paying subscription in good standing	Full access	Charges successfully	Until next billing event
<code>past_due</code>	Payment failed but still retrying	Access per grace policy	Retry attempts active	Until payment succeeds or retry limit
<code>unpaid</code>	Payment retries exhausted, subscription suspended	No access	No further retries	Until manual intervention
<code>cancelled</code>	Subscription terminated by customer or system	No access (or until period_end)	No future charges	Permanent
<code>paused</code>	Temporarily suspended by customer request	No access	No charges during pause	Until resume_date

The state transitions are governed by specific events and business rules:

Current State	Event	Next State	Action Taken	Conditions
incomplete	Payment succeeds	active or trialing	Provision access, start trial if configured	Valid payment method confirmed
incomplete	Payment fails	cancelled	Send failure notification	After maximum retry attempts
trialing	Trial expires	active	Charge first payment, continue service	Successful payment collection
trialing	Trial expires	past_due	Start dunning process	Payment collection fails
trialing	Customer cancels	cancelled	Terminate access immediately	No refund due (trial was free)
active	Renewal payment succeeds	active	Extend billing period, continue access	Normal billing cycle
active	Renewal payment fails	past_due	Start retry sequence, maintain access	Begin grace period
active	Customer cancels	cancelled or active	Schedule termination	Immediate vs end-of-period
active	Customer pauses	paused	Suspend access, stop billing	If pause feature enabled
past_due	Payment succeeds	active	Restore full access, extend period	During retry window
past_due	Retry limit exceeded	unpaid	Suspend access, stop retries	After final dunning attempt
unpaid	Payment method updated	active	Charge outstanding amount, restore access	Manual intervention successful
unpaid	Extended grace period	cancelled	Permanent termination	Business decision to close account
paused	Resume requested	active	Restore access, resume billing	Customer-initiated resume
paused	Resume date reached	active	Automatic restoration	Scheduled resume
cancelled	Reactivation requested	active	Create new subscription	Within reactivation window

Critical Design Principle: State transitions must be atomic and auditable. Every state change should be recorded with a timestamp, triggering event, and actor (system, customer, admin). This audit trail is essential for customer support, revenue recognition, and debugging billing issues.

The state machine enforces important business invariants that prevent data corruption and revenue leakage:

Access Control Invariants: Only `active` and `trialing` subscriptions grant feature access. The `past_due` state may grant continued access during a grace period, but this must be explicitly configured per business policy.

Billing Invariants: Charges only occur for `active` subscriptions or when transitioning from `trialing` to `active`. Past due subscriptions do not generate new charges - they only retry collecting existing outstanding amounts.

Cancellation Invariants: Once a subscription reaches `cancelled` state, it cannot directly transition back to `active`. Reactivation must create a new subscription entity to maintain clear audit trails and prevent accidental billing.

Temporal Invariants: State transitions must respect billing cycle boundaries. For example, a cancellation with `at_period_end=true` should not immediately change state - instead, it schedules a future transition.

ADR: Renewal Processing Architecture

The subscription billing system must process thousands of renewals daily with precise timing and reliable failure handling. This architectural decision determines how we schedule and execute recurring billing operations across the entire customer base.

Decision: Event-Driven Renewal Processing with Cron Trigger

- **Context:** We need to process subscription renewals reliably at specific dates and times, handle failures gracefully, and support both immediate and scheduled billing operations. Traditional cron-based systems can miss renewals if the system is down, while purely event-driven systems need careful orchestration for time-based triggers.
- **Options Considered:**
 1. **Pure Cron-Based Processing:** Traditional scheduled jobs that query for due renewals
 2. **Pure Event-Driven Processing:** All renewals triggered by timer events or external signals
 3. **Hybrid Event-Driven with Cron Bootstrap:** Cron triggers event publishing, events drive actual processing
- **Decision:** Hybrid approach with cron-triggered event publishing and event-driven renewal processing
- **Rationale:** Combines reliable scheduling with robust failure handling and clear separation of concerns between scheduling and processing logic
- **Consequences:** Enables retry logic, audit trails, and distributed processing while maintaining predictable renewal timing

Option	Pros	Cons	Complexity
Pure Cron	Simple implementation, predictable timing, easy debugging	No retry logic, single point of failure, difficult to scale	Low
Pure Event-Driven	Excellent failure handling, scalable, testable	Complex time-based triggering, harder to ensure coverage	High
Hybrid Approach	Best of both worlds, robust failure handling, scalable	Moderate complexity, two systems to maintain	Medium

The hybrid architecture works through a clear separation of responsibilities:

Cron Scheduler Component: Runs every hour and identifies subscriptions with `billing_cycle_anchor` dates that fall within the next processing window. For each eligible subscription, it publishes a `BillingDueEvent` to the event queue with the subscription ID and intended billing date.

Renewal Processing Engine: Consumes `BillingDueEvent` messages and executes the actual billing logic. Each event is processed independently with full retry semantics, error handling, and transaction isolation. If processing fails, the event remains in the queue for retry with exponential backoff.

Idempotency Protection: Each billing event includes an idempotency key based on the subscription ID and billing period start date. This prevents duplicate charges if the same renewal is processed multiple times due to retries or system issues.

The processing flow follows these steps:

1. **Cron Query Phase:** Every hour, query subscriptions where `next_billing_date <= now() + 1 hour` and `status IN ('active', 'trialing')`
2. **Event Publishing Phase:** For each eligible subscription, publish `BillingDueEvent{subscription_id, billing_date, idempotency_key}` to the renewal queue
3. **Processing Phase:** Renewal workers consume events and execute billing logic within database transactions
4. **Completion Phase:** Update subscription's `next_billing_date` and `last_billing_date` fields, record audit events
5. **Failure Handling Phase:** Failed events retry with exponential backoff, with alerts after multiple failures

This architecture provides several key benefits for subscription billing reliability:

Exactly-Once Processing: The combination of idempotency keys and event-driven processing ensures each billing cycle is charged exactly once, even with system failures or retries.

Graceful Failure Handling: Individual renewal failures don't impact other customers. Failed renewals automatically retry while successful ones proceed normally.

Audit Trail: Every billing event is recorded with timestamps, processing results, and failure reasons. This audit trail is crucial for customer support and financial reconciliation.

Horizontal Scaling: Multiple renewal workers can process events in parallel, allowing the system to scale renewal processing capacity independently from the scheduling component.

Dunning and Grace Period Policies

When subscription payments fail, the system enters a carefully orchestrated dunning process designed to recover revenue while maintaining positive customer relationships. Dunning management balances aggressive payment collection with customer retention, using progressively escalating communication and access restrictions.

Think of dunning like a library's approach to overdue books. The library doesn't immediately ban patrons when books are late - instead, they start with gentle email reminders, then firmer notices, and finally restrict borrowing privileges only after extended non-compliance. Throughout this process, patrons can restore good standing by simply returning books and paying modest late fees.

Subscription dunning follows similar principles but with more sophisticated timing and communication strategies. The goal is to recover failed payments while preserving long-term customer value and minimizing involuntary churn.

Grace Period Configuration: The grace period defines how long customers retain service access after payment failure. This policy varies significantly based on business model and customer segment:

Customer Segment	Grace Period	Access Policy	Rationale
Enterprise Plans	7-14 days	Full access maintained	High LTV, complex procurement processes
Annual Subscribers	5-7 days	Full access maintained	Strong commitment signal, seasonal payment issues
Monthly Consumer	3-5 days	Limited access or full restriction	Higher churn rate, faster resolution needed
Trial Conversions	1-2 days	Immediate restriction	Lower engagement, price-sensitive segment

During the grace period, the system implements a structured retry schedule that balances collection success with customer experience:

Immediate Retry (T+0): Attempt payment collection within 1 hour of initial failure. Many payment failures are temporary (insufficient funds that resolve quickly, temporary network issues) and succeed on immediate retry.

Short-Term Retries (T+1 to T+3 days): Retry payment every 24 hours during the first three days. This catches customers who need time to update payment methods or resolve banking issues.

Extended Retries (T+4 to T+7 days): Retry every 48-72 hours during the extended grace period. At this point, also escalate to email and in-app notifications requesting payment method updates.

Final Attempt (T+7 days): Make final payment attempt before transitioning subscription to `unpaid` status. Send clear communication about impending service suspension and provide easy payment recovery options.

The dunning communication strategy escalates in tone and urgency while maintaining professional, helpful messaging:

Retry Attempt	Communication Type	Message Tone	Call-to-Action
1st Failure	Email notification	Informational	"Update payment method"
2nd Attempt	Email + In-app banner	Concerned	"Avoid service interruption"
3rd Attempt	Email + Account restriction	Urgent	"Immediate action required"
Final Notice	Email + SMS (if enabled)	Final warning	"Service suspended in 24 hours"

Smart Retry Logic: Modern dunning systems implement intelligent retry timing based on failure codes from payment processors. Different failure types suggest different retry strategies:

```
Insufficient Funds (decline_code: insufficient_funds):
```

- Retry after 3 days (payday timing)
- Retry after 7 days (next payday)
- Higher success rate on Fridays and first of month

```
Card Expired (decline_code: expired_card):
```

- Immediate retry (customer may have updated)
- Email notification to update payment method
- No point in automatic retries without customer action

```
Network Error (decline_code: processing_error):
```

- Immediate retry
- Retry after 1 hour
- Treat as temporary technical issue

Revenue Recovery Optimization: Effective dunning processes can recover 15-40% of initially failed payments through strategic timing and communication. The key metrics to track include:

- **Recovery Rate:** Percentage of failed payments ultimately collected during dunning period
- **Recovery Time:** Average days between failure and successful collection
- **Involuntary Churn Rate:** Percentage of customers lost due to payment failures (vs voluntary cancellations)
- **Customer Satisfaction:** Support ticket volume and satisfaction scores during dunning periods

Dunning Automation Rules: The system should automatically adjust dunning behavior based on customer history and payment patterns:

Good Payment History: Customers with 12+ months of successful payments receive extended grace periods and gentler communication, recognizing that payment failures are likely temporary issues rather than inability to pay.

New Customers: Recent subscribers receive accelerated dunning with proactive support outreach, as payment failures often indicate setup issues rather than financial problems.

High-Value Customers: Accounts with high monthly values or annual contracts trigger immediate human intervention alongside automated dunning, ensuring personalized support for revenue-critical relationships.

Dunning Best Practice: Always provide customers with clear self-service options to resolve payment issues. Include direct links to update payment methods, view current balances, and contact support. Many customers want to resolve payment issues quickly but need easy paths to do so.

Implementation Guidance

The subscription lifecycle management system requires careful coordination between state management, payment processing, and customer communication. This implementation guidance provides the foundational code structure and core logic patterns needed to build reliable subscription billing.

Technology Recommendations:

Component	Simple Option	Advanced Option	Rationale
State Machine	Enum + switch statements	State pattern with transition validation	Start simple, refactor as complexity grows
Background Jobs	APScheduler (Python)	Celery + Redis	APScheduler sufficient for moderate scale
Event Queue	Database table + polling	Redis/RabbitMQ message queue	Database approach simpler for MVP
Retry Logic	Custom exponential backoff	Tenacity library	Tenacity provides robust retry patterns
Audit Logging	Structured logging to files	Centralized logging (ELK stack)	File logging adequate for single instance

Recommended Project Structure:

```

subscription_billing/
subscription/
    __init__.py
    models.py      # Subscription entity and state definitions
    lifecycle.py   # State machine and transition logic
    renewal_processor.py # Billing cycle processing
    dunning_manager.py # Failed payment handling
    state_machine.py  # State transition validation
billing/
    invoice_generator.py # Invoice creation (next section)
    payment_processor.py # Payment gateway integration
events/
    __init__.py
    event_publisher.py  # Event publishing infrastructure
    handlers.py        # Event handlers for lifecycle events
jobs/
    __init__.py
    renewal_scheduler.py # Cron job for identifying due renewals
    dunning_scheduler.py # Scheduled dunning operations
  
```

Core Infrastructure - Event System:

```
from enum import Enum

from datetime import datetime, timedelta

from typing import Dict, Any, Optional

from uuid import UUID

import logging

class SubscriptionEventType(Enum):

    CREATED = "subscription.created"

    ACTIVATED = "subscription.activated"

    RENEWED = "subscription.renewed"

    PAYMENT_FAILED = "subscription.payment_failed"

    CANCELLED = "subscription.cancelled"

    PAUSED = "subscription.paused"

    RESUMED = "subscription.resumed"

class SubscriptionEvent:

    def __init__(self, event_type: SubscriptionEventType, subscription_id: UUID,
                 data: Dict[str, Any], idempotency_key: Optional[str] = None):
        self.event_id = UUID.generate()
        self.event_type = event_type
        self.subscription_id = subscription_id
        self.data = data
        self.idempotency_key = idempotency_key or f"{subscription_id}_{event_type.value}_{int(datetime.utcnow().timestamp())}"
        self.created_at = datetime.utcnow()

    class EventPublisher:

        def __init__(self, database_manager: DatabaseManager):
            self.db = database_manager
```

```

self.logger = logging.getLogger(__name__)

def publish(self, event: SubscriptionEvent) -> bool:

    """Publish event to event store with idempotency protection"""

    with self.db.transaction(ISOLATION_LEVEL_SERIALIZABLE):

        # Check for duplicate events

        existing = self.db.execute(
            "SELECT event_id FROM subscription_events WHERE idempotency_key = %s",
            (event.idempotency_key,)
        ).fetchone()

        if existing:

            self.logger.info(f"Duplicate event {event.idempotency_key} ignored")

            return False

        # Insert new event

        self.db.execute("""
            INSERT INTO subscription_events (event_id, event_type, subscription_id,
                                             data, idempotency_key, created_at)
            VALUES (%s, %s, %s, %s, %s, %s)
        """, (event.event_id, event.event_type.value, event.subscription_id,
               json.dumps(event.data), event.idempotency_key, event.created_at))

    return True

```

Core Infrastructure - Audit Logging:

```
from enum import Enum

from dataclasses import dataclass

from typing import Optional, Dict, Any


class AuditEventType(Enum):

    SUBSCRIPTION_STATE_CHANGE = "subscription.state_change"

    PAYMENT_ATTEMPT = "payment.attempt"

    BILLING_CYCLE_PROCESSED = "billing.cycle_processed"

    DUNNING_ACTION = "dunning.action"


@dataclass

class AuditEvent:

    event_type: AuditEventType

    entity_id: UUID

    entity_type: str

    changes: Dict[str, Any]

    actor_id: Optional[UUID]

    metadata: Dict[str, Any]

    timestamp: datetime


class AuditLogger:

    def __init__(self, database_manager: DatabaseManager):

        self.db = database_manager


    def log_event(self, event_type: AuditEventType, entity_id: UUID,
                  entity_type: str, changes: Dict[str, Any],
                  actor_id: Optional[UUID] = None,
                  metadata: Dict[str, Any] = None) -> None:

        """Record audit event for compliance and debugging"""


```

```

audit_event = AuditEvent(
    event_type=event_type,
    entity_id=entity_id,
    entity_type=entity_type,
    changes=changes,
    actor_id=actor_id,
    metadata=metadata or {},
    timestamp=datetime.utcnow()
)

with self.db.transaction(ISOLATION_LEVEL_SERIALIZABLE):
    self.db.execute("""
        INSERT INTO audit_log (event_type, entity_id, entity_type, changes,
                             actor_id, metadata, timestamp)
        VALUES (%s, %s, %s, %s, %s, %s, %s)
    """
    , (audit_event.event_type.value, audit_event.entity_id,
       audit_event.entity_type, json.dumps(audit_event.changes),
       audit_event.actor_id, json.dumps(audit_event.metadata),
       audit_event.timestamp))

```

Core Logic Skeleton - Subscription Lifecycle Manager:

```
from enum import Enum

from datetime import datetime, timedelta

from typing import Optional, Dict, Any

from decimal import Decimal


class SubscriptionStatus(Enum):

    INCOMPLETE = "incomplete"

    TRIALING = "trialing"

    ACTIVE = "active"

    PAST_DUE = "past_due"

    UNPAID = "unpaid"

    CANCELLED = "cancelled"

    PAUSED = "paused"


class SubscriptionLifecycleManager:

    def __init__(self, database_manager: DatabaseManager,
                 event_publisher: EventPublisher,
                 audit_logger: AuditLogger):

        self.db = database_manager

        self.events = event_publisher

        self.audit = audit_logger

    def create_subscription(self, customer_id: UUID, plan_id: UUID,
                           payment_method_id: UUID, trial_end_date: Optional[datetime] = None) -> Subscription:

        """Create new subscription for customer with specified plan"""

        with self.db.transaction(ISOLATION_LEVEL_SERIALIZABLE):

            # TODO 1: Load plan details to determine trial period and billing interval
```

```

        # TODO 2: Calculate billing_cycle_anchor based on plan interval (monthly = day
of month, yearly = day of year)

        # TODO 3: Set initial status (trailing if trial period, incomplete if payment
setup needed)

        # TODO 4: Calculate next_billing_date based on trial_end_date or immediate
billing

        # TODO 5: Insert subscription record with all calculated fields

        # TODO 6: Publish SUBSCRIPTION_CREATED event with plan and customer details

        # TODO 7: Log audit event for subscription creation

        # TODO 8: Return created subscription object

        pass

def transition_state(self, subscription_id: UUID, new_status: SubscriptionStatus,
                     reason: str, actor_id: Optional[UUID] = None) -> bool:
    """Transition subscription to new state with validation and audit trail"""

    with self.db.transaction(ISOLATION_LEVEL_SERIALIZABLE):

        # TODO 1: Load current subscription with row-level lock (SELECT FOR UPDATE)

        # TODO 2: Validate state transition is allowed (use transition matrix)

        # TODO 3: Update subscription status and transition timestamp

        # TODO 4: Record state change in audit log with old/new status and reason

        # TODO 5: Publish appropriate event based on new state

        # TODO 6: Execute state-specific side effects (provision access, send
notifications)

        # Hint: Use a transition validation table to check current_state -> new_state
is valid

        pass

def process_renewal(self, subscription_id: UUID, billing_date: datetime,
                    idempotency_key: str) -> bool:
    """Process recurring billing for active subscription"""

```

```

        with self.db.transaction(ISOLATION_LEVEL_SERIALIZABLE):

            # TODO 1: Check idempotency - return True if already processed this billing
            cycle

            # TODO 2: Load subscription and verify it's in renewable state (active,
            trialing)

            # TODO 3: Calculate billing period (start_date = last_billing_date, end_date =
            next_billing_date)

            # TODO 4: Generate invoice for base plan charges (delegate to invoice
            generator)

            # TODO 5: Attempt payment collection (delegate to payment processor)

            # TODO 6: Handle payment result - update next_billing_date if successful, start
            dunning if failed

            # TODO 7: Update subscription's last_billing_date and billing cycle count

            # TODO 8: Publish SUBSCRIPTION_RENEWED event with billing details

            # Hint: Use idempotency_key = f"{subscription_id}_{billing_date.strftime('%Y-
            %m-%d')}""

        pass

    def cancel_subscription(self, subscription_id: UUID, cancel_at_period_end: bool = True,
                           reason: str = "customer_request", actor_id: Optional[UUID] = None) -> bool:
        """Cancel subscription immediately or at end of current billing period"""

        with self.db.transaction(ISOLATION_LEVEL_SERIALIZABLE):

            # TODO 1: Load subscription and verify it's cancellable (not already cancelled)

            # TODO 2: If cancel_at_period_end=False, transition to cancelled immediately

            # TODO 3: If cancel_at_period_end=True, set cancelled_at = current_period_end,
            keep status active

            # TODO 4: Record cancellation reason and schedule termination job if needed

            # TODO 5: Publish SUBSCRIPTION_CANCELLED event with cancellation details

            # TODO 6: Log audit event with cancellation reason and effective date

            # Hint: Use a separate cancelled_at field rather than changing status
            immediately

```

pass

Core Logic Skeleton - Dunning Manager:

```
from datetime import datetime, timedelta
from typing import List, Dict, Any
import time
```

```
class DunningAction(Enum):
    RETRY_PAYMENT = "retry_payment"
    SEND_EMAIL = "send_email"
    RESTRICT_ACCESS = "restrict_access"
    SUSPEND_SUBSCRIPTION = "suspend_subscription"
```

```
class DunningManager:
    def __init__(self, database_manager: DatabaseManager,
                 payment_processor: PaymentProcessor,
                 notification_service: NotificationService):
        self.db = database_manager
        self.payment_processor = payment_processor
        self.notifications = notification_service
```

```
def handle_payment_failure(self, subscription_id: UUID, invoice_id: UUID,
                           failure_reason: str) -> None:
    """Initialize dunning process for failed payment"""

    with self.db.transaction(ISOLATION_LEVEL_SERIALIZABLE):
        # TODO 1: Transition subscription to past_due status if currently active

        # TODO 2: Create dunning_attempts record with failure details and retry
        schedule

        # TODO 3: Schedule immediate retry attempt (many failures are temporary)

        # TODO 4: Send initial payment failure notification to customer

        # TODO 5: Log audit event for dunning process start

        # Hint: Store retry_count and next_retry_date in dunning_attempts table
```

PYTHON

```

    pass

def process_dunning_retry(self, subscription_id: UUID) -> bool:
    """Execute scheduled dunning retry attempt"""

    with self.db.transaction(ISOLATION_LEVEL_SERIALIZABLE):

        # TODO 1: Load subscription and outstanding invoice details

        # TODO 2: Check if still within dunning window (not exceeded max retry count)

        # TODO 3: Attempt payment collection using stored payment method

        # TODO 4: If successful: transition to active, clear dunning state, extend
        billing period

        # TODO 5: If failed: increment retry count, schedule next attempt with backoff

        # TODO 6: If retry limit exceeded: transition to unpaid status, suspend access

        # TODO 7: Send appropriate notification based on retry result

        # TODO 8: Log audit event for dunning attempt

        # Hint: Use exponential backoff - day 1, 3, 7, 14 for retry timing

    pass

    def calculate_next_retry_date(self, retry_count: int, last_attempt: datetime) ->
        datetime:

        """Calculate next retry date using exponential backoff with jitter"""

        # TODO 1: Define base retry intervals [1, 3, 7, 14] days for attempts 1-4

        # TODO 2: Add random jitter ( $\pm 2$  hours) to prevent thundering herd

        # TODO 3: Respect business hours - schedule retries during 9AM-5PM customer
        timezone

        # TODO 4: Return calculated datetime for next retry attempt

        # Hint: Use random.uniform(-2, 2) for jitter in hours

    pass

```

Milestone Checkpoint:

After implementing the subscription lifecycle management:

1. Unit Test Coverage: Run `python -m pytest subscription/test_lifecycle.py -v` and verify:

- All state transitions are validated correctly
- Renewal processing handles idempotency
- Dunning retries follow exponential backoff
- Audit events are created for all state changes

2. Integration Test: Create a test subscription and verify:

- New subscription starts in correct state (trialing or active)
- Trial expiration triggers billing attempt
- Payment failures initiate dunning process
- Successful dunning recovery restores active status
- Cancellation respects `at_period_end` setting

3. Manual Verification:

- Check database - subscription state changes should have corresponding `audit_log` entries
- Verify idempotency - processing the same renewal twice should not create duplicate charges
- Test dunning - failed payments should schedule retry attempts with correct timing

Common Issues and Debugging:

Symptom	Likely Cause	Diagnosis	Fix
Duplicate renewals	Missing idempotency check	Check <code>subscription_events</code> table for duplicate <code>idempotency_key</code>	Add unique constraint on <code>idempotency_key</code>
State transitions fail	Invalid transition attempted	Check <code>audit_log</code> for rejected transitions	Add transition validation matrix
Dunning not triggering	Cron job not running or query incorrect	Check <code>dunning_attempts</code> table creation	Verify scheduler configuration and SQL query
Payment retries too aggressive	Incorrect backoff calculation	Log <code>retry_count</code> and <code>next_retry_date</code>	Implement exponential backoff correctly

Proration and Plan Changes

Milestone(s): Milestone 3 (Proration & Plan Changes) - implements upgrade/downgrade with prorated charges and credits for mid-cycle plan changes

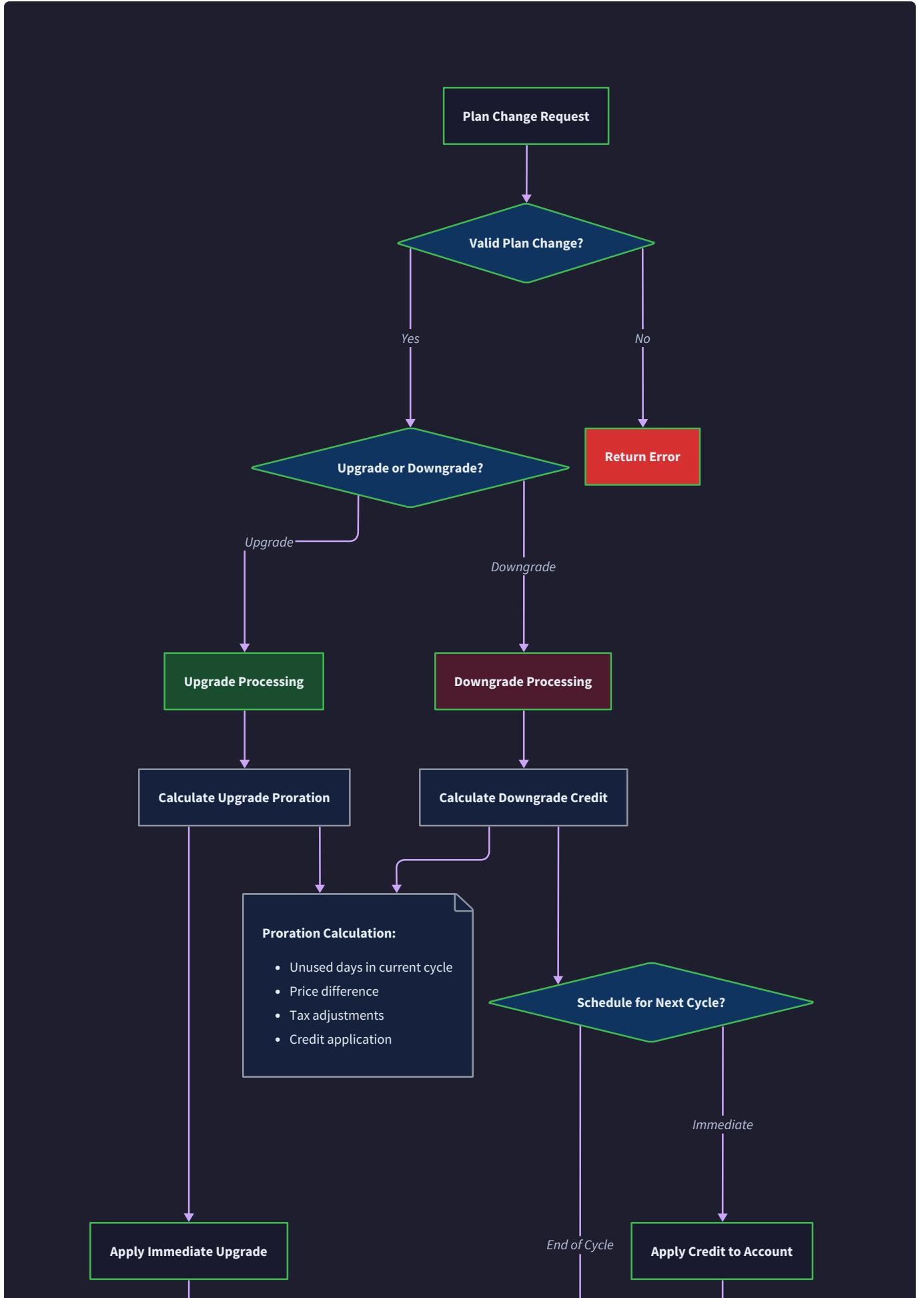
Mental Model: Partial Refunds and Charges

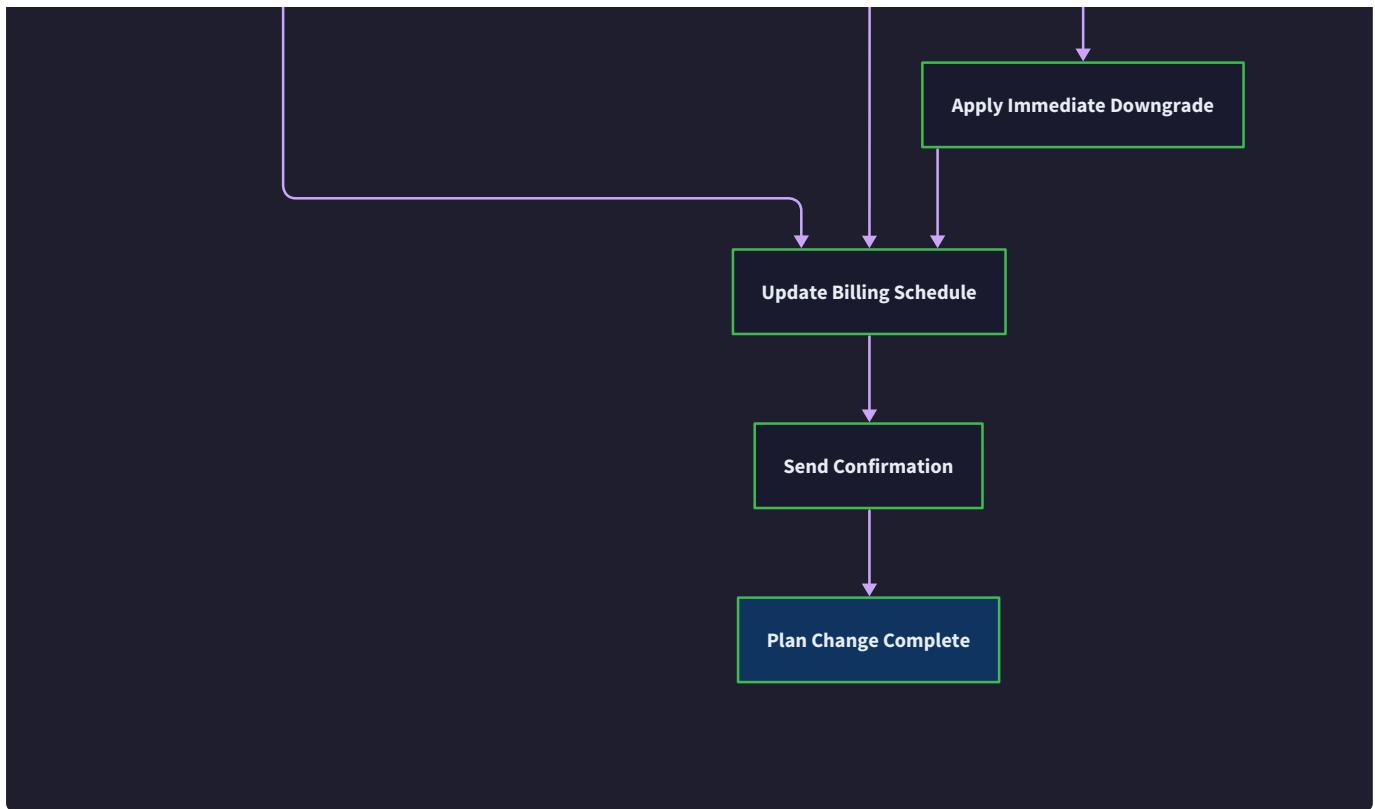
Think of proration like your monthly gym membership that you upgrade mid-month from Basic (\$30) to Premium (\$60). When you upgrade on day 15 of a 30-day billing cycle, you've already paid for the full month of Basic service, but now you want Premium for the remaining 15 days. The gym needs to figure out: "How much extra should we charge for the upgrade?" They calculate that you used half your Basic membership (15 days worth = \$15), so you have \$15 of unused Basic time. Premium costs \$60, so for 15 days it's \$30. The additional charge is \$30 - \$15 = \$15.

This same logic applies when you downgrade. If you switch from Premium to Basic mid-cycle, you've overpaid and deserve a credit for the difference. The gym calculates how much Premium time you have left, converts it to Basic pricing, and either refunds the difference or applies it as credit to your account.

Proration is this fair calculation system that ensures customers pay exactly for what they use, even when they change plans mid-cycle. It prevents both customer frustration ("Why am I paying for features I'm not using?") and business revenue leakage ("Why are we giving away premium features for basic pricing?").

The mathematical challenge is handling edge cases: What happens during February with 28 days when your billing cycle started on a 31-day month? What about timezone differences when a customer upgrades at 11:59 PM? How do you round fractions of cents fairly across thousands of customers? These details can make or break the fairness and accuracy of your billing system.





Proration Calculation Algorithms

Proration calculations form the mathematical foundation for fair billing when customers change their subscription plans mid-cycle. The core principle is proportional allocation: customers should pay for exactly the service they receive, regardless of when they make changes during their billing period.

Time-Based Proration Formula

Time-based proration calculates charges based on the remaining days in the current billing cycle. This is the most common proration method for subscription billing systems.

Variable	Description	Example Value
days_remaining	Days left in current billing cycle	15
total_cycle_days	Total days in current billing period	30
old_plan_price	Monthly price of current plan	\$30.00
new_plan_price	Monthly price of target plan	\$60.00
usage_factor	Fraction of cycle remaining	0.5 (15/30)

The proration algorithm follows these steps:

1. **Calculate the usage factor** by dividing remaining days by total cycle days. This gives the proportion of the billing cycle that will use the new plan.

2. **Determine unused value from the old plan** by multiplying the old plan price by the usage factor. This represents the monetary value of service time the customer has already paid for but won't receive.
3. **Calculate new plan cost for remaining period** by multiplying the new plan price by the usage factor. This represents the cost of providing the new service level for the remaining time.
4. **Compute the proration amount** as the difference between new plan cost and unused old plan value. Positive values indicate additional charges; negative values indicate credits owed.
5. **Handle currency precision** by rounding to the smallest currency unit (cents) using consistent rounding rules across all calculations.

Key Insight: The proration calculation must be deterministic and repeatable. Running the same calculation multiple times with identical inputs must always produce the same result, even if processed days apart.

Quantity-Based Proration Formula

Quantity-based proration applies when plan changes involve user seats, storage limits, or other measurable quantities rather than time periods.

Variable	Description	Example Value
old_quantity	Current quantity (seats, GB, etc.)	5 seats
new_quantity	Target quantity after change	8 seats
unit_price	Price per unit for remaining period	\$10/seat
days_remaining	Days left in billing cycle	15
total_cycle_days	Total days in billing period	30

Quantity-based proration follows this algorithm:

1. **Calculate the quantity difference** between new and old quantities. This determines how many additional units require billing.
2. **Determine the time factor** using the same method as time-based proration (remaining days divided by total cycle days).
3. **Calculate unit cost for remaining period** by multiplying the unit price by the time factor.
4. **Compute total proration charge** by multiplying the quantity difference by the unit cost for the remaining period.
5. **Apply minimum and maximum quantity constraints** as defined by the plan rules to ensure the change is within allowed boundaries.

Combined Proration Scenarios

Real-world plan changes often involve both time and quantity adjustments simultaneously. For example, upgrading from a Basic plan (5 seats, \$50/month) to a Professional plan (unlimited seats, \$100/month) mid-cycle requires calculating both the base plan change and any quantity adjustments.

The combined algorithm processes these changes in sequence:

1. **Calculate base plan proration** using time-based formulas for the plan-level pricing difference.
2. **Calculate quantity adjustments** using quantity-based formulas for any seat or limit changes.
3. **Sum all proration components** to determine the total adjustment amount.
4. **Validate the total amount** against business rules such as maximum single-transaction limits or credit balance caps.

Currency and Precision Handling

Proration calculations must maintain financial precision to prevent accumulated errors across thousands of transactions. The system stores all monetary values in the smallest currency unit (cents for USD) and performs all calculations using integer arithmetic.

Precision Rule	Description	Implementation
Currency Storage	Store amounts as integers in smallest unit	<code>amount_cents: int</code>
Division Rounding	Use consistent rounding for fractional cents	Round half up ($0.5 \rightarrow 1$)
Calculation Order	Perform multiplication before division	<code>(price * days) / total_days</code>
Result Validation	Verify totals match expected ranges	Check against plan min/max prices

Critical Warning: Floating-point arithmetic can introduce rounding errors that accumulate across many customers. Always use integer arithmetic with the smallest currency unit to maintain exact precision in financial calculations.

ADR: Credit Balance Management

Decision: Customer Credit Balance Architecture

Context: When customers downgrade plans or receive refunds from proration calculations, the system must decide how to handle credit amounts. Credits could be applied immediately to reduce current invoices, stored as account balance for future use, or processed as direct refunds to the customer's payment method.

Options Considered:

- Immediate Credit Application:** Credits automatically reduce the next invoice amount
- Account Credit Balance:** Credits accumulate in customer account for flexible future use
- Direct Refund Processing:** Credits trigger immediate refunds to original payment method

Decision: Implement account credit balance with automatic application to future invoices

Rationale: Account credit balances provide the optimal balance between customer flexibility and operational simplicity. Immediate application works only when invoices are pending, missing opportunities for mid-cycle credits. Direct refunds create payment processing overhead and potential fraud concerns with frequent small refunds. Account balances handle all timing scenarios while maintaining customer transparency.

Consequences: Requires additional database storage for credit tracking, audit trail for credit movements, and customer portal visibility into credit balance. Enables superior customer experience with flexible credit usage and simplified customer service operations.

Option	Pros	Cons
Immediate Credit Application	Simple implementation, no additional storage	Only works with pending invoices, loses mid-cycle credits
Account Credit Balance	Flexible timing, handles all scenarios, customer-friendly	Requires credit management system, additional complexity
Direct Refund Processing	Immediate customer satisfaction, clean accounting	Payment processing fees, fraud risks, operational overhead

The credit balance architecture implements these components:

Credit Balance Storage: Each `Customer` entity includes a `credit_balance_cents` field that tracks the accumulated credit amount. This balance persists across billing cycles and plan changes, providing continuity for customers with varying usage patterns.

Credit Transaction Logging: Every credit application, whether from proration calculations, manual adjustments, or promotional offers, generates an `AuditEvent` record with complete transaction details. This

creates an immutable audit trail for financial reconciliation and customer service inquiries.

Automatic Credit Application: During invoice generation, the billing engine automatically applies available credit balance to reduce the invoice total. Credits apply before payment processing, reducing the amount charged to the customer's payment method.

Credit Balance Limits: The system enforces maximum credit balance limits per customer to prevent abuse and manage financial exposure. Excess credits beyond the limit trigger alternative handling such as direct refunds or promotional extensions.

Credit Application Algorithm

The credit application process follows these steps during invoice generation:

1. **Retrieve customer credit balance** from the database with row-level locking to prevent concurrent modifications during the application process.
2. **Calculate applicable credit amount** as the minimum of available credit balance and outstanding invoice amount. This ensures credits never create negative invoice totals.
3. **Update customer credit balance** by subtracting the applied amount using atomic database operations to maintain consistency.
4. **Apply credit to invoice** by adding a credit line item that reduces the total amount due before payment processing.
5. **Log credit application transaction** with complete details for audit trail and customer visibility.
6. **Handle remaining balance** by either charging the customer for any remaining invoice amount or noting full payment via credit if the balance covers the entire invoice.

Common Proration Pitfalls

Proration calculations contain numerous edge cases that can cause billing errors, customer complaints, and revenue discrepancies. Understanding these pitfalls helps developers build robust billing systems that handle real-world complexity gracefully.

⚠ Pitfall: Calendar Month Variations

Description: Calculating proration based on "days in month" without considering calendar variations leads to inconsistent billing amounts. A customer who upgrades on January 15th (31-day month) receives different proration than one who upgrades on February 15th (28-day month), even though both upgrade mid-cycle.

Why It's Wrong: Customers expect consistent proration logic regardless of calendar quirks. Variable month lengths create apparent unfairness where identical timing produces different charges.

How to Fix: Use billing cycle anchor dates rather than calendar months. If a customer's billing cycle runs from the 15th to the 14th each month, always calculate proration based on that specific cycle period, regardless of underlying calendar days.

Pitfall: Timezone Confusion in Proration Timing

Description: Using server timezone or customer timezone inconsistently when determining proration effective dates. A customer who upgrades at 11:59 PM in their timezone might be processed as upgrading the next day in server timezone, affecting proration calculations.

Why It's Wrong: Proration amounts can change significantly based on which day the system considers the "upgrade date." This creates unpredictable billing behavior and customer service issues.

How to Fix: Always use the customer's account timezone for billing calculations and store the timezone-aware timestamp with each plan change event. Convert to UTC for database storage but perform all billing logic in customer timezone.

Pitfall: Double-Counting Credits During Multiple Plan Changes

Description: When customers make multiple plan changes within a single billing cycle, each change calculation might include credits from previous changes, leading to compound credit errors.

Why It's Wrong: Customers receive excessive credits that reduce revenue and create accounting discrepancies. Multiple plan changes should calculate proration based on actual service periods, not overlapping credit calculations.

How to Fix: Track plan change effective dates precisely and calculate proration only for the actual service period under each plan. Use a sequential approach where each plan change starts from the effective date of the previous change.

Pitfall: Rounding Errors Accumulating Across Customers

Description: Inconsistent rounding of fractional cents across proration calculations creates systematic bias that can add up to significant amounts across thousands of customers.

Why It's Wrong: Even small rounding inconsistencies can result in meaningful revenue differences when applied across a large customer base. Regulatory and accounting requirements demand consistent financial precision.

How to Fix: Implement standardized rounding rules (typically "round half up" for financial calculations) and apply them consistently across all proration scenarios. Document the rounding approach and verify it meets local financial regulations.

Pitfall: Proration on Trial Periods

Description: Applying proration calculations to customers who are still in trial periods, especially when trials have zero cost. The system might attempt to calculate credits or charges for "upgrading" from a free trial.

Why It's Wrong: Trial periods should not generate proration charges since customers haven't paid for the trial service. Proration logic designed for paid plans doesn't apply to trial scenarios.

How to Fix: Check subscription status before applying proration logic. For trial subscriptions, treat plan changes as immediate switches without proration. Begin proration calculations only after the trial period ends.

and paid billing begins.

⚠ Pitfall: Currency Conversion During Proration

Description: Performing currency conversion before or during proration calculations when customers change plans that involve different currencies. Exchange rate fluctuations can affect proration accuracy.

Why It's Wrong: Currency conversion rates change continuously, making proration calculations non-deterministic. The same plan change processed at different times might yield different amounts due to exchange rate variations.

How to Fix: Perform all proration calculations in the customer's account currency without conversion. If plan changes involve currency switches, treat them as separate operations: complete proration in the original currency, then apply the currency change for future billing cycles.

⚠ Pitfall: Negative Credit Balances

Description: Allowing customer credit balances to become negative due to calculation errors, failed credit applications, or manual adjustments without proper validation.

Why It's Wrong: Negative credit balances represent debt owed by customers but tracked in the credit system rather than through normal invoice processes. This creates accounting confusion and collection challenges.

How to Fix: Implement credit balance constraints that prevent negative values. When operations would result in negative credits, handle them through the regular billing process by generating invoices for the additional amounts owed.

Implementation Guidance

The proration and credit management system requires careful implementation to handle the mathematical complexity and edge cases inherent in mid-cycle billing adjustments. This implementation focuses on accuracy, auditability, and performance for high-volume billing operations.

Technology Recommendations

Component	Simple Option	Advanced Option
Calculation Engine	Python Decimal with basic rounding	Custom financial arithmetic library
Credit Storage	PostgreSQL with ACID transactions	Event-sourced credit ledger
Currency Handling	Single currency with integer cents	Multi-currency with exchange rate API
Audit Logging	Database audit table	Dedicated audit service with immutable log
Proration Rules	Static configuration in code	Dynamic rule engine with business rules

Recommended File Structure

```
subscription-billing/
  src/
    billing/
      proration/
        __init__.py
        calculator.py      ← Core proration calculation engine
        credit_manager.py   ← Customer credit balance management
        plan_change_handler.py ← Orchestrates plan changes with proration
        exceptions.py       ← Proration-specific error types
        models.py           ← Proration data structures
    core/
      money.py            ← Money type with currency handling
      audit.py             ← Audit event logging
      database.py          ← Database manager
  tests/
    test_proration.py    ← Proration calculation tests
    test_credit_manager.py ← Credit management tests
    test_plan_changes.py  ← End-to-end plan change tests
```

Core Money and Currency Infrastructure

```
# billing/core/money.py                                         PYTHON

from decimal import Decimal, ROUND_HALF_UP
from typing import Optional
import re

DECIMAL_PRECISION = Decimal('0.01')

DEFAULT_CURRENCY = 'USD'

SUPPORTED_CURRENCIES = {

    'USD': {'symbol': '$', 'decimal_places': 2, 'minor_unit': 100},
    'EUR': {'symbol': '€', 'decimal_places': 2, 'minor_unit': 100},
    'GBP': {'symbol': '£', 'decimal_places': 2, 'minor_unit': 100},
}

class Money:

    """Immutable money type that maintains financial precision."""

    def __init__(self, amount_cents: int, currency_code: str):

        if not isinstance(amount_cents, int):
            raise ValueError("Amount must be an integer representing cents")

        if currency_code not in SUPPORTED_CURRENCIES:
            raise ValueError(f"Unsupported currency: {currency_code}")

        self.amount_cents = amount_cents
        self.currency_code = currency_code

    @classmethod
```

```
def from_decimal(cls, amount: Decimal, currency: str) -> 'Money':  
    """Create Money from decimal amount with proper rounding."""  
  
    if currency not in SUPPORTED_CURRENCIES:  
        raise ValueError(f"Unsupported currency: {currency}")  
  
  
    minor_unit = SUPPORTED_CURRENCIES[currency]['minor_unit']  
  
    cents = int((amount * minor_unit).quantize(Decimal('1'), ROUND_HALF_UP))  
  
    return cls(cents, currency)  
  
  
  
def to_decimal(self) -> Decimal:  
    """Convert Money to decimal representation."""  
  
    minor_unit = SUPPORTED_CURRENCIES[self.currency_code]['minor_unit']  
  
    return Decimal(self.amount_cents) / minor_unit  
  
  
  
def __add__(self, other: 'Money') -> 'Money':  
    if self.currency_code != other.currency_code:  
        raise ValueError("Cannot add money with different currencies")  
  
    return Money(self.amount_cents + other.amount_cents, self.currency_code)  
  
  
  
def __sub__(self, other: 'Money') -> 'Money':  
    if self.currency_code != other.currency_code:  
        raise ValueError("Cannot subtract money with different currencies")  
  
    return Money(self.amount_cents - other.amount_cents, self.currency_code)  
  
  
  
def __mul__(self, factor: Decimal) -> 'Money':  
    """Multiply money by decimal factor with proper rounding."""
```

```
    result_cents = (Decimal(self.amount_cents) * factor).quantize(Decimal('1'),
ROUND_HALF_UP)

    return Money(int(result_cents), self.currency_code)

def __eq__(self, other) -> bool:

    if not isinstance(other, Money):

        return False

    return (self.amount_cents == other.amount_cents and

            self.currency_code == other.currency_code)
```

Proration Calculation Engine Skeleton

```
# billing/proration/calculator.py                                PYTHON

from decimal import Decimal

from datetime import datetime, date

from typing import Dict, List, Optional

from dataclasses import dataclass

from ..core.money import Money, DECIMAL_PRECISION

from .models import ProrationResult, ProrationContext

from .exceptions import ProrationError


@dataclass

class ProrationResult:

    """Result of a proration calculation with detailed breakdown."""

    credit_amount: Money

    charge_amount: Money

    net_amount: Money # positive = charge, negative = credit

    calculation_details: Dict

    effective_date: datetime


class ProrationCalculator:

    """Calculates prorated charges and credits for plan changes."""

    def calculate_plan_change_proration(

        self,

        old_plan: 'Plan',

        new_plan: 'Plan',

        change_date: datetime,
```

```
        billing_cycle_start: datetime,  
  
        billing_cycle_end: datetime  
  
) -> ProrationResult:  
  
    """Calculate proration for mid-cycle plan changes.  
  
    Args:
```

```
        old_plan: Current subscription plan  
  
        new_plan: Target plan after change  
  
        change_date: When the change becomes effective  
  
        billing_cycle_start: Start of current billing period  
  
        billing_cycle_end: End of current billing period
```

```
Returns:
```

```
    ProrationResult with charge/credit amounts and calculation details
```

```
"""
```

```
# TODO 1: Validate that change_date falls within the current billing cycle
```

```
#         Raise ProrationError if change_date is outside cycle boundaries
```

```
# TODO 2: Calculate total days in billing cycle
```

```
#         Use (billing_cycle_end - billing_cycle_start).days
```

```
#         Handle timezone-aware datetime objects properly
```

```
# TODO 3: Calculate days remaining from change_date to cycle end
```

```
#         Use (billing_cycle_end - change_date).days
```

```
#         Ensure result is not negative
```

```

# TODO 4: Calculate usage factor as Decimal(days_remaining) /
Decimal(total_cycle_days)

#           This represents the fraction of cycle using the new plan


# TODO 5: Calculate unused value from old plan

#           unused_value = old_plan.base_price * usage_factor

#           This is credit owed for unused time on old plan


# TODO 6: Calculate new plan cost for remaining period

#           new_plan_cost = new_plan.base_price * usage_factor

#           This is charge for new plan during remaining time


# TODO 7: Calculate net proration amount

#           net_amount = new_plan_cost - unused_value

#           Positive = customer owes money, Negative = customer gets credit


# TODO 8: Create ProrationResult with calculated amounts and details dict

#           Include calculation breakdown for audit trail and customer transparency


# Hint: All Money arithmetic should use the currency from old_plan

# Hint: Store intermediate calculations in calculation_details dict

pass


def calculate_quantity_proration(
    self,
    plan: 'Plan',
    old_quantity: int,

```

```
new_quantity: int,  
  
change_date: datetime,  
  
billing_cycle_start: datetime,  
  
billing_cycle_end: datetime  
  
) -> ProrationResult:  
  
    """Calculate proration for quantity changes (seats, storage, etc.)."""  
  
    # TODO 1: Validate quantity change parameters  
  
    # Ensure new_quantity meets plan minimum and maximum limits  
  
    # Verify change_date is within billing cycle  
  
  
    # TODO 2: Calculate quantity difference (new_quantity - old_quantity)  
  
    # This can be positive (increase) or negative (decrease)  
  
  
    # TODO 3: Calculate time factor using same logic as plan change proration  
  
    # time_factor = days_remaining / total_cycle_days  
  
  
    # TODO 4: Get per-unit price for the plan  
  
    # Check if plan uses per-seat, per-GB, or other unit pricing  
  
  
    # TODO 5: Calculate unit cost for remaining period  
  
    # unit_cost = per_unit_price * time_factor  
  
  
    # TODO 6: Calculate total proration amount  
  
    # total_amount = quantity_difference * unit_cost  
  
  
    # TODO 7: Create ProrationResult with quantity-based calculation details  
  
    # Include unit pricing and quantity change information
```

pass

Credit Balance Manager Skeleton

```
# billing/proration/credit_manager.py                                PYTHON

from typing import Optional, List

from uuid import UUID

from datetime import datetime

from ..core.money import Money

from ..core.audit import AuditEventType

from ..core.database import DatabaseManager


class CreditBalanceManager:

    """Manages customer credit balances and applications."""

    def __init__(self, db_manager: DatabaseManager):

        self.db = db_manager


    def apply_credit(
            self,
            customer_id: UUID,
            amount: Money,
            source: str,
            reference_id: Optional[UUID] = None,
            actor_id: Optional[UUID] = None
        ) -> bool:

        """Add credit to customer's account balance.

        Args:
            customer_id: Customer receiving credit
        """

        pass
```

```
    amount: Credit amount to add

    source: Description of credit source (e.g., "plan_downgrade", "refund")

    reference_id: Related transaction ID for audit trail

    actor_id: ID of user/system making the change
```

Returns:

```
    True if credit applied successfully
```

```
"""
```

```
# TODO 1: Start database transaction with SERIALIZABLE isolation
```

```
#     Use self.db.transaction(ISOLATION_LEVEL_SERIALIZABLE)
```

```
# TODO 2: Lock customer record for update to prevent concurrent modifications
```

```
#     SELECT customer credit_balance_cents FOR UPDATE WHERE customer_id = ?
```

```
# TODO 3: Validate credit amount is positive
```

```
#     Raise ValueError if amount is negative or zero
```

```
# TODO 4: Update customer credit balance
```

```
#     new_balance = current_balance + credit_amount
```

```
#     UPDATE customers SET credit_balance_cents = ? WHERE customer_id = ?
```

```
# TODO 5: Create audit event for credit application
```

```
#     Use AuditEventType.CREDIT_APPLIED with amount and source details
```

```
#     Call self.log_event() with complete transaction details
```

```
# TODO 6: Commit transaction and return success status
```

```
#     Handle database errors and rollback on failure
```

```
pass

def consume_credit(
    self,
    customer_id: UUID,
    amount_needed: Money,
    actor_id: Optional[UUID] = None
) -> Money:

    """Apply customer credit to reduce an amount owed.

    Args:
```

Args:

```
    customer_id: Customer using credit

    amount_needed: Total amount that could be reduced by credit

    actor_id: ID of user/system applying credit
```

Returns:

```
    Money representing actual credit amount applied (may be less than requested)

    """

# TODO 1: Start database transaction with proper isolation

#         Lock customer record to prevent concurrent credit usage

# TODO 2: Get current customer credit balance

#         SELECT credit_balance_cents FROM customers WHERE customer_id = ?

# TODO 3: Calculate applicable credit amount

#         applied_credit = min(available_credit, amount_needed)

#         Cannot apply more credit than customer has or needs
```

```
# TODO 4: Update customer credit balance

#           new_balance = current_balance - applied_credit

#           Ensure balance never goes negative


# TODO 5: Create audit event for credit consumption

#           Log the credit usage with before/after balances


# TODO 6: Return the actual credit amount applied

#           This may be less than amount_needed if insufficient credit

pass


def get_credit_balance(self, customer_id: UUID) -> Money:

    """Get current credit balance for customer."""

    # TODO 1: Query customer credit balance from database

    #           Handle case where customer doesn't exist


    # TODO 2: Convert stored cents to Money object

    #           Use customer's default currency from account settings

pass
```

Plan Change Handler Skeleton

```
# billing/proration/plan_change_handler.py                                PYTHON

from uuid import UUID

from datetime import datetime

from typing import Optional


from .calculator import ProrationCalculator

from .credit_manager import CreditBalanceManager

from ..core.money import Money

from .exceptions import ProrationError, PlanChangeError


class PlanChangeHandler:

    """Orchestrates plan changes with proration calculations."""

    def __init__(self, proration_calc: ProrationCalculator, credit_manager: CreditBalanceManager):

        self.proration_calc = proration_calc

        self.credit_manager = credit_manager


    def process_plan_upgrade(
            self,
            subscription_id: UUID,
            new_plan_id: UUID,
            effective_date: Optional[datetime] = None,
            actor_id: Optional[UUID] = None
        ) -> Dict:

        """Process subscription upgrade with proration charges."""

        # TODO 1: Load subscription and validate current state
```

```
#           Ensure subscription is active and not cancelled

#           Verify new_plan_id exists and is available for customer

# TODO 2: Load current and target plans

#           Validate that target plan is actually an upgrade (higher tier/price)

#           Check plan compatibility and upgrade path rules

# TODO 3: Determine effective date for change

#           Default to current datetime if not specified

#           Validate effective_date is not in the past

# TODO 4: Calculate proration using ProrationCalculator

#           Get current billing cycle boundaries

#           Call calculate_plan_change_proration with all parameters

# TODO 5: Process proration charges

#           If net_amount > 0, create invoice for additional charge

#           If net_amount < 0, apply credit to customer account

# TODO 6: Update subscription with new plan

#           Change subscription.plan_id to new_plan_id

#           Record plan change event with effective date

# TODO 7: Return plan change summary

#           Include proration details, new plan info, and effective date

pass
```

```
def process_plan_downgrade(  
    self,  
    subscription_id: UUID,  
    new_plan_id: UUID,  
    effective_date: Optional[datetime] = None,  
    actor_id: Optional[UUID] = None  
) -> Dict:  
  
    """Process subscription downgrade with proration credits."""  
  
    # TODO 1: Load and validate subscription and plans similar to upgrade  
  
    # Verify new plan is actually a downgrade  
  
    # TODO 2: Calculate proration (will typically result in credit)  
  
    # Use same ProrationCalculator.calculate_plan_change_proration  
  
    # TODO 3: Apply credit to customer account  
  
    # Use CreditBalanceManager.apply_credit for proration amount  
  
    # Record credit source as "plan_downgrade"  
  
    # TODO 4: Handle feature access changes  
  
    # Immediately restrict access to features not in new plan  
  
    # Update feature entitlements in subscription record  
  
    # TODO 5: Update subscription and log downgrade event  
  
    # Record reason for downgrade if provided  
  
    pass
```

Milestone Checkpoint

After implementing the proration system, verify functionality with these tests:

Unit Tests: Run `python -m pytest tests/test_proration.py -v` Expected output should show all proration calculation tests passing, including edge cases for calendar months, timezone handling, and currency precision.

Integration Tests: Test complete plan change workflows:

1. Create subscription with monthly plan (\$30/month)
2. Upgrade to premium plan (\$60/month) on day 15 of 30-day cycle
3. Verify proration charge of \$15 (half month difference)
4. Check customer credit balance remains unchanged
5. Confirm subscription reflects new plan and features

Manual Verification:

- Use billing admin interface to trigger plan change
- Verify proration calculation shows detailed breakdown
- Confirm audit events recorded for all balance changes
- Test edge cases: month boundaries, leap years, timezone differences

Common Issues to Check:

- Proration amounts should always be deterministic for same inputs
- Currency precision should maintain cent accuracy
- Credit balances should never become negative without explicit business logic
- All plan changes should generate complete audit trails

Usage-Based Billing Engine

Milestone(s): Milestone 4 (Usage-Based Billing) - implements metered billing with usage tracking, aggregation, and reporting

Mental Model: Utility Metering

Think of usage-based billing like your electricity meter at home. Every time you turn on a light, run the dishwasher, or charge your laptop, the meter records consumption. At the end of the month, the utility company reads your meter, calculates your total usage, applies their tiered pricing structure (first 500 kWh at \$0.10, next 500 kWh at \$0.12, etc.), and generates your bill.

The usage-based billing engine works exactly the same way. Instead of measuring kilowatt-hours, we might be measuring API calls, storage gigabytes, or video transcoding minutes. Instead of a physical meter, we have

software that captures usage events. Instead of a meter reader visiting your house, we have automated aggregation processes that sum up consumption. And instead of utility rate schedules, we have pricing tiers defined in our subscription plans.

Just like utility billing, the key challenges are accuracy (every usage event must be counted exactly once), timing (usage must be attributed to the correct billing period), and pricing complexity (different rates for different usage levels). The meter never lies, and neither should our usage tracking system.

This mental model helps us understand why certain architectural decisions matter. Just as utility companies need redundant meter readings and audit trails for regulatory compliance, we need idempotent event processing and complete usage records. Just as utilities batch-process millions of meter readings efficiently, we need scalable aggregation systems. And just as your electricity bill shows detailed breakdowns of usage and charges, our invoicing system must clearly explain usage-based fees to customers.

Usage Event Ingestion

The foundation of any usage-based billing system is reliable event ingestion. Usage events represent discrete measurable activities that customers perform within your service. These might be API requests, data processing jobs, storage operations, or any other billable activity. The ingestion system must handle high volumes of events while ensuring perfect accuracy and preventing duplicate charges.

Event Structure and Validation

Every usage event follows a standardized structure that captures the essential billing information. The event must identify the customer, specify what type of activity occurred, record the quantity consumed, and timestamp when the activity took place. Additional metadata helps with debugging and provides context for customer support scenarios.

Field	Type	Description
event_id	UUID	Globally unique identifier for this specific event
customer_id	UUID	Foreign key linking to the customer who performed this activity
subscription_id	UUID	Optional foreign key to the specific subscription being billed
event_type	str	Categorizes the type of billable activity (e.g., "api_call", "storage_gb_hour")
quantity	Decimal	Precise measurement of consumption using arbitrary precision arithmetic
timestamp	datetime	Server-side timestamp when the event was recorded (UTC)
idempotency_key	str	Client-provided key for preventing duplicate event submission
metadata	Dict	Additional context like user agent, IP address, or feature flags
source_service	str	Which internal service generated this event for debugging

The ingestion API validates every field before accepting events. Customer IDs must reference existing customers. Event types must match configured billable metrics. Quantities must be positive numbers with reasonable precision limits. Timestamps must fall within acceptable ranges (not too far in the past or future). Metadata must conform to size limits to prevent database bloat.

Idempotency and Deduplication Strategies

The most critical requirement for usage event ingestion is preventing duplicate charges. Network retries, client-side bugs, and distributed system failures can all cause the same usage event to be submitted multiple times. Charging customers twice for the same API call or storage operation destroys trust and creates billing disputes.

Decision: Client-Provided Idempotency Keys

- **Context:** Usage events may be submitted multiple times due to network failures, client retries, or distributed system issues
- **Options Considered:** Server-generated deduplication, client-provided idempotency keys, event fingerprinting
- **Decision:** Require clients to provide idempotency keys with event submission
- **Rationale:** Client-provided keys give the event source control over deduplication semantics and work across service restarts
- **Consequences:** Clients must implement idempotency key generation, but we get reliable duplicate prevention

Option	Pros	Cons
Server-generated deduplication	Simple client implementation	Cannot handle retries across service restarts
Client-provided idempotency keys	Reliable across all failure modes	Requires client-side key management
Event fingerprinting	Transparent to clients	Complex edge cases with similar events

The idempotency key should be deterministic based on the usage activity itself. For example, an API request might use a combination of request ID, endpoint, and timestamp. A file upload might use the file hash and user ID. The key must uniquely identify the specific billable activity, not just the client request.

When the ingestion service receives an event, it first checks if an event with the same idempotency key already exists. If found, it returns success without creating a duplicate record. If not found, it inserts the new event atomically. This check-and-insert operation must be atomic to handle concurrent submissions of the same event.

Event Processing Pipeline

Usage events flow through a multi-stage processing pipeline that validates, enriches, and prepares them for aggregation. Each stage has specific responsibilities and error handling requirements.

The validation stage ensures data quality and business rule compliance. It verifies that the customer has an active subscription that includes the billable metric. It checks that the event timestamp falls within acceptable bounds. It validates that the quantity is reasonable for the event type (e.g., API calls should have quantity 1, storage events might have fractional quantities).

The enrichment stage adds derived fields that simplify downstream processing. It calculates which billing period the event belongs to based on the customer's subscription cycle. It looks up pricing information for the event type. It adds geographic or regulatory metadata that affects billing calculations. It resolves the subscription plan version that was active when the event occurred.

The normalization stage converts events into a standardized format for aggregation. It handles unit conversions (e.g., bytes to gigabytes, milliseconds to hours). It applies any business logic for event grouping or categorization. It adds partition keys that optimize aggregation queries. It ensures consistent precision for decimal quantities.

Real-Time vs Batch Processing

Usage events can be processed immediately as they arrive or collected into batches for bulk processing. Each approach has different trade-offs for latency, throughput, and resource utilization.

Real-time processing provides immediate feedback to customers about their usage and remaining quotas. It enables real-time rate limiting and usage alerts. It distributes processing load evenly over time rather than creating periodic spikes. However, it requires more complex infrastructure to handle traffic bursts and may have higher per-event overhead.

Batch processing achieves higher throughput by amortizing processing costs across many events. It simplifies error handling and retry logic. It enables more sophisticated aggregation algorithms that consider the entire batch. However, it introduces processing delays and creates periodic resource spikes that must be provisioned for.

Most production systems use a hybrid approach where high-priority events (quotas, alerts) are processed in real-time while bulk aggregation happens in periodic batches. This provides responsive user experience while maintaining processing efficiency.

Error Handling and Event Recovery

Usage event ingestion must handle various failure scenarios gracefully while preserving data integrity.

Network timeouts, database connection failures, and validation errors all require different recovery strategies.

Transient failures like network timeouts or temporary database unavailability should trigger automatic retries with exponential backoff. The retry logic must preserve idempotency by using the same idempotency key across all attempts. Clients should also implement retries with the same idempotency key to ensure end-to-end reliability.

Validation errors like invalid customer IDs or malformed event types should be logged and returned immediately to the client. These events should not be retried automatically since they will continue to fail until the client fixes the data. However, they should be stored in a dead letter queue for debugging and potential manual recovery.

Processing failures during enrichment or normalization stages should be retried since they often involve temporary issues with external services. Failed events should be quarantined and processed separately to avoid blocking the healthy event stream. Processing delays should be monitored and alerted since they can affect billing accuracy.

ADR: Usage Aggregation Architecture

Usage events must be aggregated into billing periods to calculate charges for each customer. The aggregation system processes potentially millions of individual events and produces summary records that feed into invoice generation. The architecture must balance processing efficiency, data consistency, and query performance.

Decision: Batch Aggregation with Real-Time Approximation

- **Context:** Usage events arrive continuously but billing calculations happen monthly, requiring efficient aggregation of large event volumes
- **Options Considered:** Pure real-time aggregation, pure batch aggregation, hybrid batch + real-time approximation
- **Decision:** Primary batch aggregation with real-time approximation for customer dashboards and quota enforcement
- **Rationale:** Batch processing handles billing-scale volumes efficiently while real-time approximation provides responsive user experience
- **Consequences:** More complex architecture but optimal for both billing accuracy and user experience

Option	Pros	Cons
Pure real-time aggregation	Immediate accuracy, simple architecture	Poor performance at billing scale, expensive infrastructure
Pure batch aggregation	Excellent throughput, cost-effective	Poor user experience, delayed quota enforcement
Hybrid batch + real-time	Best of both approaches	Complex architecture, data consistency challenges

Batch Aggregation Design

The primary aggregation system processes usage events in scheduled batches, typically every few hours or daily. This approach maximizes throughput by processing large numbers of events together using optimized database queries and parallel processing techniques.

The batch processor groups events by customer, subscription, event type, and billing period. It calculates sum, count, maximum, and other aggregate functions depending on the billable metric. For example, API call events are summed to get total calls, while concurrent user events might use maximum values to determine peak usage.

Aggregation windows align with billing period boundaries to ensure accurate monthly totals. The processor maintains checkpoint records to track which events have been aggregated, enabling incremental processing and recovery from failures. It handles events that arrive late (after their billing period has ended) by updating historical aggregations and triggering billing corrections if necessary.

The output of batch aggregation feeds directly into invoice generation. These records represent the official usage totals used for billing calculations and must be completely accurate. They include detailed breakdowns by event type and time period to support customer inquiries and billing disputes.

Real-Time Approximation System

A separate real-time system provides approximate usage totals for customer dashboards and quota enforcement. This system prioritizes responsiveness over perfect accuracy, using techniques like sampling, estimation, and eventual consistency.

Real-time aggregation maintains rolling windows of recent usage events using in-memory data structures. It provides fast lookups for current billing period totals and usage trends. The data may be slightly stale or approximate, but it updates within seconds of new events arriving.

For quota enforcement, the real-time system errs on the side of caution by overestimating usage when in doubt. This prevents customers from accidentally exceeding their limits while the approximation catches up. Periodic reconciliation with batch aggregation results corrects any overestimations.

Customer-facing dashboards clearly indicate when they're showing estimated vs final usage totals. During the billing period, customers see real-time estimates. After billing closes, they see the final calculated amounts that appear on their invoices.

Data Consistency and Reconciliation

The hybrid architecture introduces potential inconsistencies between real-time approximations and batch aggregation results. The system must detect and resolve these discrepancies to maintain data integrity.

Reconciliation jobs compare real-time and batch totals for completed billing periods. Small differences (within expected tolerances) are logged but not acted upon. Larger discrepancies trigger investigation workflows that examine the underlying event data to identify the source of disagreement.

When inconsistencies are detected, the batch aggregation results take precedence for billing purposes since they process complete event sets with full consistency guarantees. The real-time system updates its models

and estimates based on the authoritative batch results.

Historical usage data remains immutable once billing closes to ensure audit trail integrity. Any corrections due to late-arriving events or processing errors are recorded as separate adjustment records rather than modifying original totals.

Overage and Tiered Usage Billing

Most usage-based pricing models include base allowances with overage charges when customers exceed their plan limits. These pricing structures require sophisticated calculation logic that considers plan entitlements, usage tiers, and billing period boundaries.

Usage Allowances and Quota Management

Each subscription plan defines base allowances for various usage metrics. For example, a "Professional" plan might include 10,000 API calls, 100 GB of storage, and 50 hours of video transcoding per month. These allowances are included in the base subscription fee without additional charges.

Plan Component	Type	Description
base_allowance	int	Included quantity for this usage metric
overage_rate_cents	int	Per-unit charge when usage exceeds allowance
usage_metric	str	Specific billable activity this allowance covers
reset_frequency	BillingInterval	How often the allowance resets (monthly, annually)
soft_limit_threshold	float	Percentage of allowance that triggers usage warnings
hard_limit_enabled	bool	Whether to block usage when allowance is exhausted

The system tracks current usage against these allowances throughout the billing period. When customers approach their limits (typically at 80% and 95% thresholds), the system sends usage alert notifications. These alerts help customers manage their consumption and avoid unexpected overage charges.

Some plans implement hard limits that prevent additional usage once allowances are exhausted. Others allow unlimited overage with per-unit billing. The quota management system enforces these policies in real-time using the approximation data while ensuring accurate billing using the batch aggregation results.

Tiered Overage Pricing

Overage charges often use tiered pricing structures where different usage levels have different per-unit rates. This mirrors utility billing models where higher consumption levels have higher marginal costs.

For example, API call overages might be priced as:

- First 5,000 overage calls: \$0.002 per call
- Next 20,000 overage calls: \$0.0015 per call

- Additional overage calls: \$0.001 per call

The calculation engine processes usage through each tier sequentially. If a customer uses 30,000 overage calls, they pay the first-tier rate for the first 5,000 calls, the second-tier rate for the next 20,000 calls, and the third-tier rate for the remaining 5,000 calls.

Tier Definition	Field	Type	Description
tier_number	int	Sequential tier ordering (1, 2, 3, ...)	
min_quantity	int	Minimum usage level for this tier (inclusive)	
max_quantity	Optional[int]	Maximum usage level for this tier (None for unlimited)	
price_per_unit_cents	int	Per-unit charge for usage in this tier	
flat_fee_cents	int	One-time charge when entering this tier	

The tiered calculation logic must handle edge cases like partial tier usage, multiple billing periods, and plan changes mid-cycle. It maintains precision using integer arithmetic and the smallest currency units to avoid floating-point rounding errors.

Cross-Metric Dependencies and Bundling

Some pricing models include dependencies between different usage metrics. For example, a plan might bundle API calls and storage together, or provide discounts when customers use multiple services simultaneously.

Bundled allowances share usage quotas across multiple metrics. A customer might have "10,000 combined API calls and webhook deliveries" rather than separate allowances for each. The system tracks total usage across all bundled metrics and applies overage charges when the combined total exceeds the bundle allowance.

Tiered discounts apply different rates based on total spending or usage volume across all metrics. High-volume customers might receive better overage rates as they move into higher spending tiers. These calculations require aggregate usage data across all billable metrics for the customer.

Cross-metric calculations add complexity to the aggregation system since it must consider usage relationships rather than processing each metric independently. The calculation engine evaluates these dependencies after individual metric aggregation but before final invoice generation.

Billing Period Boundary Handling

Usage allowances and overage calculations must account for billing period boundaries, especially when customers change plans mid-cycle or when events arrive after billing has closed.

Plan changes during a billing period require prorated allowance calculations. If a customer upgrades from a plan with 5,000 API call allowance to a plan with 15,000 allowance halfway through the month, they get 5,000

$+ (15,000 / 2) = 12,500$ total allowance for that billing period. The overage calculation uses the higher tier rates for the entire period.

Late-arriving events can affect overage calculations for closed billing periods. If events arrive after invoices are generated, the system must determine whether to issue credit adjustments or carry the usage forward to the next billing period. Most systems apply a grace period (e.g., 72 hours) for late events and generate credit adjustments for material changes.

Multi-period aggregation handles customers with annual plans or custom billing cycles. The system maintains running totals across the entire subscription period while providing monthly breakdowns for customer reporting. Allowance resets and overage calculations align with the customer's specific billing cycle rather than calendar months.

Usage-Based Pricing Psychology

Effective usage-based pricing considers customer psychology and billing predictability. Customers prefer pricing models they can understand and budget for, even if those models are more expensive than purely usage-based alternatives.

Generous base allowances reduce billing anxiety by covering typical usage patterns without overage charges. Most customers should fall within their plan allowances most months. This makes the subscription feel predictable while still allowing monetization of high-usage customers through overage fees.

Clear usage monitoring and alerts help customers manage their consumption proactively. Real-time dashboards show current usage, remaining allowances, and projected monthly totals. Usage trend analysis helps customers understand their consumption patterns and choose appropriate plans.

Graduated overage pricing with reasonable first-tier rates reduces sticker shock when customers first exceed their allowances. The highest overage rates should only apply to extremely high usage levels where customers clearly derive significant value from the service.

Implementation Guidance

The usage-based billing engine requires careful attention to data precision, event reliability, and scalable aggregation. The following implementation provides a foundation for accurate usage tracking while maintaining high throughput and low latency for customer-facing features.

Technology Recommendations

Component	Simple Option	Advanced Option
Event Ingestion	HTTP REST API with JSON	Apache Kafka with schema registry
Event Storage	PostgreSQL with partitioning	Apache Cassandra or ClickHouse
Real-time Aggregation	Redis with rolling windows	Apache Flink or Kafka Streams
Batch Processing	Python with pandas	Apache Spark or Airflow
Metrics and Monitoring	Prometheus with Grafana	DataDog or New Relic
Queue Management	PostgreSQL-based queues	Redis Streams or Amazon SQS

Recommended File Structure

```

subscription-billing/
  usage/
    __init__.py
    models.py      ← UsageEvent, PricingTier, UsageAggregation
    ingestion.py   ← Event ingestion API and validation
    aggregation.py ← Batch aggregation engine
    realtime.py    ← Real-time approximation system
    pricing.py     ← Overage and tiered pricing calculations
    quota.py       ← Quota enforcement and usage limits
    events.py      ← Event publishing for usage alerts
  tests/
    test_usage_ingestion.py
    test_aggregation.py
    test_pricing_tiers.py

```

Core Data Models

```
from decimal import Decimal, ROUND_HALF_UP

from datetime import datetime, timezone

from typing import Optional, Dict, Any

import uuid

from dataclasses import dataclass

from enum import Enum


class UsageMetricType(Enum):

    API_CALLS = "api_calls"

    STORAGE_GB_HOURS = "storage_gb_hours"

    BANDWIDTH_GB = "bandwidth_gb"

    TRANSCODING_MINUTES = "transcoding_minutes"

    ACTIVE_USERS = "active_users"


@dataclass

class UsageEvent:

    """Single usage event representing billable activity."""

    event_id: uuid.UUID

    customer_id: uuid.UUID

    subscription_id: Optional[uuid.UUID]

    event_type: str

    quantity: Decimal

    timestamp: datetime

    idempotency_key: str

    metadata: Dict[str, Any]

    source_service: str

    billing_period_start: Optional[datetime] = None

    billing_period_end: Optional[datetime] = None
```

```
processed_at: Optional[datetime] = None
```



```
def __post_init__(self):
```

```
    if self.timestamp.tzinfo is None:
```

```
        self.timestamp = self.timestamp.replace(tzinfo=timezone.utc)
```



```
@dataclass
```

```
class PricingTier:
```

```
    """Usage pricing tier with quantity ranges and rates."""
```

```
    tier_number: int
```

```
    min_quantity: int
```

```
    max_quantity: Optional[int] # None for unlimited
```

```
    price_per_unit_cents: int
```

```
    flat_fee_cents: int = 0
```



```
def contains_quantity(self, quantity: int) -> bool:
```

```
    if quantity < self.min_quantity:
```

```
        return False
```

```
    if self.max_quantity is not None and quantity > self.max_quantity:
```

```
        return False
```

```
    return True
```



```
def calculate_tier_charge(self, quantity: int) -> int:
```

```
    """Calculate total charge for quantity within this tier."""
```

```
    if not self.contains_quantity(quantity):
```

```
        raise ValueError(f"Quantity {quantity} not in tier range")
```

```
tier_quantity = min(quantity, self.max_quantity or quantity) - self.min_quantity +  
1  
  
return self.flat_fee_cents + (tier_quantity * self.price_per_unit_cents)  
  
@dataclass  
  
class UsageAggregation:  
  
    """Aggregated usage totals for billing period."""  
  
    aggregation_id: uuid.UUID  
  
    customer_id: uuid.UUID  
  
    subscription_id: uuid.UUID  
  
    event_type: str  
  
    billing_period_start: datetime  
  
    billing_period_end: datetime  
  
    total_quantity: Decimal  
  
    event_count: int  
  
    first_event_at: datetime  
  
    last_event_at: datetime  
  
    created_at: datetime  
  
  
    def to_billing_quantity(self) -> int:  
  
        """Convert decimal quantity to integer for billing calculations."""  
  
        return int(self.total_quantity.quantize(Decimal('1'), rounding=ROUND_HALF_UP))
```

Usage Event Ingestion Service

```
from typing import List, Optional
```

PYTHON

```
import logging
```

```
from datetime import datetime, timezone
```

```
from decimal import Decimal, InvalidOperation
```

```
logger = logging.getLogger(__name__)
```

```
class UsageIngestionService:
```

```
    """Service for receiving and validating usage events."""
```

```
def __init__(self, db_manager, event_publisher, subscription_service):
```

```
    self.db = db_manager
```

```
    self.events = event_publisher
```

```
    self.subscriptions = subscription_service
```

```
def submit_usage_event(self, event_data: Dict[str, Any]) -> UsageEvent:
```

```
    """
```

```
        Submit a single usage event with idempotency protection.
```

```
        Returns existing event if idempotency key matches.
```

```
    """
```

```
# TODO 1: Validate event_data contains required fields
```

```
# TODO 2: Check for existing event with same idempotency_key
```

```
# TODO 3: Validate customer_id exists and has active subscription
```

```
# TODO 4: Validate event_type is supported billable metric
```

```
# TODO 5: Validate quantity is positive decimal with reasonable precision
```

```
# TODO 6: Enrich event with billing period information
```

```
# TODO 7: Insert event record atomically with idempotency check
```

```
# TODO 8: Publish event for real-time aggregation
```

```
# TODO 9: Return created or existing event
pass

def submit_usage_batch(self, events: List[Dict[str, Any]]) -> List[UsageEvent]:
    """Submit multiple usage events in a single transaction."""

    # TODO 1: Validate all events in batch before processing any

    # TODO 2: Group events by idempotency key to detect duplicates within batch

    # TODO 3: Start database transaction with appropriate isolation level

    # TODO 4: Process each event using submit_usage_event logic

    # TODO 5: Collect results and rollback transaction on any failure

    # TODO 6: Publish batch event notification for monitoring

    # TODO 7: Return list of created/existing events

    pass

def _validate_event_data(self, event_data: Dict[str, Any]) -> None:
    """Validate event data structure and business rules."""

    required_fields = ['customer_id', 'event_type', 'quantity', 'idempotency_key']

    for field in required_fields:

        if field not in event_data:

            raise ValueError(f"Missing required field: {field}")

    # Validate UUID fields

    try:

        uuid.UUID(event_data['customer_id'])

        if 'subscription_id' in event_data:

            uuid.UUID(event_data['subscription_id'])

    except (ValueError, TypeError) as e:
```

```
        raise ValueError(f"Invalid UUID format: {e}")

    # Validate quantity

    try:

        quantity = Decimal(str(event_data['quantity']))

        if quantity <= 0:

            raise ValueError("Quantity must be positive")

        if quantity.as_tuple().exponent < -6:  # Max 6 decimal places

            raise ValueError("Quantity precision too high")

    except (InvalidOperation, TypeError):

        raise ValueError("Invalid quantity format")



def _enrich_event_data(self, event_data: Dict[str, Any]) -> Dict[str, Any]:
    """Add derived fields for downstream processing."""

    enriched = event_data.copy()

    # Add server timestamp if not provided

    if 'timestamp' not in enriched:

        enriched['timestamp'] = datetime.now(timezone.utc)

    # TODO: Look up billing period boundaries from subscription

    # TODO: Add pricing tier information for this event type

    # TODO: Add customer segment or region for regulatory requirements

    # TODO: Calculate partition key for efficient storage

    return enriched
```

Usage Aggregation Engine

```
from collections import defaultdict
from typing import Dict, List, Tuple
import logging

logger = logging.getLogger(__name__)

class UsageAggregationEngine:

    """Batch processing engine for usage event aggregation."""

    def __init__(self, db_manager, metrics_client):
        self.db = db_manager
        self.metrics = metrics_client
        self._aggregation_functions = {
            'sum': self._sum_aggregation,
            'count': self._count_aggregation,
            'max': self._max_aggregation,
            'last_value': self._last_value_aggregation
        }

    def aggregate_billing_period(self,
                                 billing_period_start: datetime,
                                 billing_period_end: datetime) -> List[UsageAggregation]:
        """
        Aggregate all usage events for the specified billing period.

        Returns list of aggregation records for invoice generation.
        """

        # TODO 1: Query all usage events in billing period range
        # TODO 2: Group events by (customer_id, subscription_id, event_type)
        pass
```

```
# TODO 3: Calculate aggregations for each group using appropriate function

# TODO 4: Create UsageAggregation records with calculated totals

# TODO 5: Store aggregation records atomically

# TODO 6: Update aggregation checkpoint for incremental processing

# TODO 7: Publish aggregation completed events

# TODO 8: Return list of aggregation records

pass

def process_incremental_aggregation(self, checkpoint_timestamp: datetime) -> int:

    """Process events since last checkpoint for real-time updates."""

    # TODO 1: Query events newer than checkpoint_timestamp

    # TODO 2: Group events and calculate incremental aggregations

    # TODO 3: Update existing aggregation records or create new ones

    # TODO 4: Handle late-arriving events for closed billing periods

    # TODO 5: Update checkpoint timestamp atomically

    # TODO 6: Return count of processed events

    pass

def _group_events_for_aggregation(self, events: List[UsageEvent]) -> Dict[Tuple,
List[UsageEvent]]:

    """Group events by aggregation key (customer, subscription, event_type)."""

    groups = defaultdict(list)

    for event in events:

        key = (event.customer_id, event.subscription_id, event.event_type)

        groups[key].append(event)

    return dict(groups)
```

```
def _sum_aggregation(self, events: List[UsageEvent]) -> Decimal:
    """Sum all event quantities."""
    return sum(event.quantity for event in events)

def _count_aggregation(self, events: List[UsageEvent]) -> Decimal:
    """Count number of events."""
    return Decimal(len(events))

def _max_aggregation(self, events: List[UsageEvent]) -> Decimal:
    """Return maximum quantity across all events."""
    return max(event.quantity for event in events) if events else Decimal('0')

def _last_value_aggregation(self, events: List[UsageEvent]) -> Decimal:
    """Return quantity from most recent event."""
    if not events:
        return Decimal('0')
    latest_event = max(events, key=lambda e: e.timestamp)
    return latest_event.quantity
```

Tiered Pricing Calculator

```
from typing import List
```

PYTHON

```
from decimal import Decimal
```

```
import logging
```

```
logger = logging.getLogger(__name__)
```

```
class TieredPricingCalculator:
```

```
    """Calculates usage charges using tiered pricing models."""
```

```
    def __init__(self, db_manager):
```

```
        self.db = db_manager
```

```
    def calculate_usage_charge(self,
```

```
                           customer_id: uuid.UUID,
```

```
                           event_type: str,
```

```
                           total_quantity: int,
```

```
                           base_allowance: int = 0) -> Dict[str, Any]:
```

```
        """
```

```
        Calculate tiered usage charge for quantity beyond base allowance.
```

```
        Returns detailed charge breakdown with tier-by-tier calculations.
```

```
        """
```

```
# TODO 1: Load pricing tiers for this event_type from plan
```

```
# TODO 2: Calculate overage quantity (total - base_allowance)
```

```
# TODO 3: Apply tiered pricing calculation to overage quantity
```

```
# TODO 4: Build detailed breakdown of charges by tier
```

```
# TODO 5: Calculate total charge amount with proper rounding
```

```
# TODO 6: Return charge details for invoice line items
```

```
        pass
```

```
def _apply_tiered_pricing(self, quantity: int, tiers: List[PricingTier]) -> Dict[str, Any]:\n\n    """Apply tiered pricing structure to calculate total charges."""\n\n    if quantity <= 0:\n\n        return {\n\n            'total_charge_cents': 0,\n\n            'tier_breakdown': [],\n\n            'effective_rate_cents': 0\n\n        }\n\n    \n\n    total_charge = 0\n\n    tier_breakdown = []\n\n    remaining_quantity = quantity\n\n    \n\n    for tier in sorted(tiers, key=lambda t: t.tier_number):\n\n        if remaining_quantity <= 0:\n\n            break\n\n        \n\n        # Calculate quantity that falls in this tier\n\n        tier_min = tier.min_quantity\n\n        tier_max = tier.max_quantity or float('inf')\n\n        tier_quantity = min(remaining_quantity, tier_max - tier_min + 1)\n\n        \n\n        if tier_quantity > 0:\n\n            # Calculate charge for this tier\n\n            tier_charge = tier.flat_fee_cents + (tier_quantity *\n            tier.price_per_unit_cents)\n\n            total_charge += tier_charge\n\n            tier_breakdown.append({\n                'tier_name': tier.name,\n                'tier_min': tier_min,\n                'tier_max': tier_max,\n                'tier_quantity': tier_quantity,\n                'tier_charge_cents': tier_charge\n            })\n\n            remaining_quantity -= tier_quantity\n\n    \n\n    return {\n        'total_charge_cents': total_charge,\n        'tier_breakdown': tier_breakdown\n    }
```

```
total_charge += tier_charge

tier_breakdown.append({
    'tier_number': tier.tier_number,
    'quantity': tier_quantity,
    'rate_cents': tier.price_per_unit_cents,
    'flat_fee_cents': tier.flat_fee_cents,
    'tier_charge_cents': tier_charge
})

remaining_quantity -= tier_quantity

effective_rate = total_charge / quantity if quantity > 0 else 0

return {
    'total_charge_cents': total_charge,
    'tier_breakdown': tier_breakdown,
    'effective_rate_cents': round(effective_rate, 2)
}
```

Real-Time Usage Approximation

```
import redis

from datetime import datetime, timedelta

from typing import Dict, Optional

import json

import logging

logger = logging.getLogger(__name__)

class RealTimeUsageTracker:

    """Provides approximate real-time usage totals for dashboards and quotas."""

    def __init__(self, redis_client: redis.Redis, window_hours: int = 24):

        self.redis = redis_client

        self.window_hours = window_hours

        self.window_seconds = window_hours * 3600

    def record_usage_event(self, event: UsageEvent) -> None:

        """Record usage event in real-time approximation system."""

        # TODO 1: Create time-bucketed key for this customer/event_type

        # TODO 2: Add event quantity to running total for current time bucket

        # TODO 3: Set expiration on time bucket keys for automatic cleanup

        # TODO 4: Update customer's current billing period total

        # TODO 5: Check if customer is approaching usage quotas

        # TODO 6: Publish usage alert if quota thresholds exceeded

        pass

    def get_current_usage(self,
                          customer_id: uuid.UUID,
```

PYTHON

```

        event_type: str,
        billing_period_start: datetime) -> Dict[str, Any]:
    """Get approximate current usage for customer billing period."""

    # TODO 1: Calculate time buckets covering billing period

    # TODO 2: Sum usage across all relevant time buckets

    # TODO 3: Apply smoothing factor for late-arriving events

    # TODO 4: Return usage estimate with confidence interval

    pass


def check_usage_quotas(self, customer_id: uuid.UUID) -> List[Dict[str, Any]]:
    """Check if customer is approaching or exceeding usage quotas."""

    # TODO 1: Get current usage estimates for all event types

    # TODO 2: Load plan allowances and quota limits

    # TODO 3: Calculate usage percentage for each metric

    # TODO 4: Identify metrics approaching warning thresholds

    # TODO 5: Return list of quota status for each metric

    pass


def _get_time_bucket_key(self, customer_id: uuid.UUID, event_type: str, timestamp: datetime) -> str:
    """Generate Redis key for time-bucketed usage data."""

    bucket_hour = timestamp.replace(minute=0, second=0, microsecond=0)

    return f"usage:{customer_id}:{event_type}:{bucket_hour.isoformat()}"

```

Milestone Checkpoints

After implementing the usage-based billing engine, verify the following functionality:

- 1. Event Ingestion Validation:** Submit duplicate events with the same idempotency key. Verify only one event is recorded and the API returns success for both submissions.

2. **Usage Aggregation Accuracy:** Submit a known set of usage events spanning multiple days. Run batch aggregation and verify the totals match expected calculations.
3. **Tiered Pricing Calculations:** Create a plan with tiered overage pricing (e.g., first 1000 units at \$0.01, next 5000 at \$0.008). Submit usage that spans multiple tiers and verify charge calculations.
4. **Real-Time Approximation:** Submit usage events and verify they appear in real-time usage dashboards within seconds. Compare real-time totals to batch aggregation results for accuracy.
5. **Quota Enforcement:** Configure usage limits and submit events that approach quotas. Verify usage alerts are sent and hard limits are enforced when configured.

Expected behavior: Usage events should be processed idempotently with perfect accuracy in batch mode and approximate accuracy in real-time mode. Pricing calculations should handle complex tiered structures without rounding errors. The system should scale to thousands of events per second while maintaining data consistency.

Invoicing and Payment Processing

Milestone(s): Foundation for all milestones - generates invoices from subscription charges, usage, and credits, then processes payments through the payment gateway integration

Mental Model: Restaurant Check Generation and Payment

Think of invoice generation like a restaurant preparing your final check at the end of a meal. The waiter (Invoice Generation Engine) walks through your table's activity: the base meal price (subscription charges), any extra appetizers you ordered during dinner (usage overages), and any comp drinks the manager gave you (credit applications). They itemize everything on a single receipt with clear line items, calculate tax, and present a total amount due.

The payment processing is like the credit card terminal at your table. Once you approve the charge, the terminal (Payment Gateway Integration) communicates with your bank behind the scenes. The restaurant doesn't handle your banking directly - they just send the charge amount and receive back a simple "approved" or "declined" response. If your card is declined, the system might automatically retry with a different payment method you have on file.

This separation of concerns - invoice generation versus payment processing - mirrors how billing systems work. The invoice engine focuses on accurately calculating what's owed based on subscription activity, while the payment processor handles the mechanics of actually collecting money through external financial networks.

Invoice Generation Engine

The **Invoice Generation Engine** serves as the financial consolidation point for all subscription-related charges, credits, and adjustments. This component transforms raw billing data from subscriptions, usage events, and proration calculations into structured invoices that customers can understand and payment systems can process.

The invoice generation process follows a specific sequence that ensures mathematical accuracy and maintains a complete audit trail. The engine operates on a **billing period boundary** principle, where it collects all chargeable events that occurred within a defined time window, applies any outstanding credits, and produces a net amount due. This approach prevents double-billing and ensures that every cent is accounted for across billing cycles.

Invoice Line Item Calculation

Each invoice contains multiple line items that represent different types of charges and credits. The engine processes these in a specific order to ensure consistent application of business rules:

Line Item Type	Source	Calculation Method	Example
Base Subscription	Plan recurring charge	Fixed amount from plan definition	\$29.99/month Pro Plan
Usage Overage	Aggregated usage events	Tiered pricing beyond allowance	1,000 API calls × \$0.01 = \$10.00
Plan Change Proration	Mid-cycle upgrades	Prorated charge for remaining days	\$15.00 upgrade × 15/30 days = \$7.50
Credit Application	Customer credit balance	Applied against positive charges	-\$5.00 account credit
Tax Calculation	Jurisdiction rules	Applied to taxable line items	\$4.20 sales tax (10%)

The engine implements **financial precision** by performing all calculations in the smallest currency unit (cents for USD) to avoid floating-point rounding errors. Each line item stores both the calculation method and the raw inputs used, enabling complete auditability of every charge.

Design Insight: Invoice line items are immutable once generated. If an error is discovered after invoice creation, the system generates a credit note (negative invoice) rather than modifying the original invoice. This maintains financial audit integrity and matches accounting best practices.

Invoice Status State Machine

Invoices progress through a well-defined lifecycle that coordinates with payment processing and dunning management:

Current Status	Trigger Event	Next Status	Actions Taken
<code>draft</code>	Complete line item calculation	<code>open</code>	Mark invoice as ready for payment
<code>open</code>	Payment received successfully	<code>paid</code>	Update customer balance, mark subscription current
<code>open</code>	Payment attempt failed	<code>past_due</code>	Start dunning process, set retry schedule
<code>open</code>	Invoice voided by admin	<code>void</code>	Cancel payment attempts, apply credit
<code>past_due</code>	Payment retry succeeded	<code>paid</code>	Complete payment flow, stop dunning
<code>past_due</code>	All retries exhausted	<code>uncollectible</code>	Suspend subscription, write off balance
<code>paid</code>	Refund processed	<code>refunded</code>	Update customer credit balance

The invoice status directly influences subscription state transitions. A subscription cannot remain `active` with `past_due` invoices beyond the configured grace period. This tight coupling ensures that billing status accurately reflects service access.

Invoice Generation Algorithms

The core invoice generation algorithm processes multiple data sources and applies business rules to produce accurate billing:

- Billing Period Determination:** Calculate the start and end dates for the current billing cycle based on the subscription's billing anchor date and interval
- Base Charge Calculation:** Retrieve the subscription's current plan and calculate the recurring charge for the billing period
- Usage Event Aggregation:** Sum all usage events for the customer within the billing period, grouped by event type
- Usage Charge Calculation:** Apply the plan's usage tiers to determine overage charges beyond included allowances
- Proration Processing:** Include any mid-cycle plan changes or subscription modifications that occurred during the billing period
- Credit Application:** Apply available customer credits to reduce the total amount due
- Tax Calculation:** Apply jurisdiction-specific tax rules to taxable line items
- Invoice Finalization:** Lock the invoice calculations, assign a sequential invoice number, and transition to `open` status
- Payment Initiation:** Trigger payment processing for the final invoice amount
- Audit Trail Recording:** Log all calculation steps and data sources for compliance and debugging

Each step in this algorithm includes validation checks to ensure data consistency. For example, the usage event aggregation step verifies that all events fall within the billing period boundaries and that no duplicate events are processed.

Common Invoice Generation Pitfalls

⚠ Pitfall: Timezone Confusion in Billing Period Boundaries

Many developers incorrectly calculate billing periods by using server local time instead of the customer's billing timezone. This can cause usage events to be attributed to the wrong billing cycle, leading to under-billing or over-billing. Always store billing anchor dates in UTC but perform billing period calculations in the customer's configured timezone, then convert event timestamps accordingly.

⚠ Pitfall: Floating Point Currency Calculations

Using floating-point arithmetic for monetary calculations introduces rounding errors that compound over many invoices. A \$0.001 rounding error across 100,000 customers results in \$100 of unaccounted variance. Always perform calculations in integer cents and use the `Money` type's `from_decimal()` method for conversions.

⚠ Pitfall: Credit Application Order Dependency

Applying customer credits before calculating taxes can result in incorrect tax amounts, while applying credits after taxes can reduce the customer's available credit balance unnecessarily. The correct order is: calculate base charges, calculate taxes, then apply credits to the total amount including taxes.

⚠ Pitfall: Invoice Immutability Violations

Modifying invoice line items after the invoice status changes to `open` breaks audit trails and can create payment inconsistencies. If an error is discovered, generate a separate credit invoice rather than modifying the original. This preserves the historical record and follows standard accounting practices.

Payment Gateway Integration

The **Payment Gateway Integration** component provides a clean abstraction layer between the subscription billing system and external payment processors. This integration handles payment method storage, charge processing, webhook management, and failure recovery while maintaining PCI compliance boundaries.

The integration operates on an **asynchronous payment model** where payment initiation and completion are separate operations. When an invoice becomes due, the system initiates a payment request with the gateway and receives an immediate response indicating whether the request was accepted for processing. The actual payment result arrives later via webhook notifications, which update the invoice status and trigger downstream actions.

Payment Method Management

Customer payment methods are stored and managed entirely within the payment gateway's secure vault to maintain PCI compliance. The billing system only stores tokenized references that can be used to initiate charges without handling sensitive payment data:

Payment Method Attribute	Storage Location	Purpose	Example Value
Gateway Token ID	Billing system database	Reference for charge requests	pm_1ABC123xyz
Payment Type	Billing system database	Display and routing logic	card, bank_account, wallet
Last Four Digits	Billing system database	Customer identification	4242
Expiration Date	Billing system database	Proactive replacement	12/2025
Is Default	Billing system database	Primary payment selection	true
Gateway Customer ID	Billing system database	Customer grouping	cus_ABC123

The system supports **payment method hierarchy** where customers can configure primary and backup payment methods. If the primary method fails, the system automatically retries with backup methods before entering the dunning process.

Design Insight: The billing system never stores raw payment information - not even temporarily. All payment data flows directly from the customer's browser to the payment gateway using tokenization. This architectural decision simplifies PCI compliance and reduces security risk.

Charge Processing Flow

Payment charges follow a standardized flow that handles both synchronous and asynchronous response patterns:

- 1. Invoice Payment Trigger:** Invoice generation completes and triggers payment processing for the final amount due
- 2. Payment Method Selection:** Select the customer's default payment method, or fall back to backup methods if primary fails
- 3. Idempotency Key Generation:** Generate a unique key combining customer ID, invoice ID, and attempt number to prevent duplicate charges
- 4. Gateway Charge Request:** Submit charge request to payment gateway with amount, payment method token, and metadata
- 5. Immediate Response Handling:** Process gateway's immediate response (accepted, declined, requires_action, processing)

6. **Status Update:** Update invoice status based on immediate response (remains `open` for async processing)
7. **Webhook Processing:** Receive asynchronous webhook with final charge result
8. **Invoice Completion:** Update invoice to `paid` or `past_due` based on final charge result
9. **Subscription Impact:** Update subscription status and access levels based on payment outcome
10. **Notification Dispatch:** Send customer notifications for payment success or failure

The charge processing includes **automatic retry logic** for transient failures. Network timeouts, gateway maintenance, and temporary declines trigger retry attempts with exponential backoff before marking the payment as failed.

Webhook Event Processing

Payment gateway webhooks provide authoritative updates about charge status, payment method changes, and dispute notifications. The webhook processing system ensures reliable event handling even under high load or temporary system outages:

Webhook Event Type	Triggered By	System Response	Example Action
<code>payment_intent.succeeded</code>	Successful charge completion	Update invoice to <code>paid</code> status	Activate subscription access
<code>payment_intent.payment_failed</code>	Charge failure after retries	Update invoice to <code>past_due</code>	Start dunning sequence
<code>payment_method.updated</code>	Customer updates card details	Refresh stored payment metadata	Update expiration date display
<code>invoice.payment_action_required</code>	3D Secure authentication needed	Notify customer of required action	Send authentication link
<code>chargeback.created</code>	Bank dispute initiated	Freeze related subscription	Suspend service, gather evidence

Webhook processing implements **exactly-once delivery semantics** using idempotency keys stored in the webhook payload. The system deduplicates webhook events and ensures that each event is processed only once, even if the gateway sends duplicate notifications.

Critical Design Decision: Webhook processing is separated from real-time payment flows. Webhooks update system state asynchronously, while immediate payment responses handle user experience. This separation ensures that gateway latency doesn't impact customer-facing operations.

ADR: Payment Webhook Processing

Decision: Asynchronous Webhook Processing with Event Queue

- **Context:** Payment gateways send webhook notifications for charge status updates, but these webhooks can arrive out of order, be duplicated, or fail during processing. The system must reliably process these events to maintain accurate billing state while handling high webhook volumes without blocking user-facing operations.
- **Options Considered:**
 1. Synchronous webhook processing with immediate database updates
 2. Asynchronous processing with message queue and worker processes
 3. Hybrid approach with immediate critical updates and queued background processing
- **Decision:** Implement fully asynchronous webhook processing using a reliable message queue with dead letter handling and idempotency protection
- **Rationale:** Payment webhooks are not latency-sensitive and require reliable processing more than immediate processing. Queue-based processing provides natural retry mechanisms, handles traffic spikes, and prevents webhook failures from impacting payment gateway relationships. The slight delay in status updates is acceptable since customers receive immediate feedback from the synchronous payment initiation response.
- **Consequences:** Enables horizontal scaling of webhook processing, provides built-in retry and dead letter handling, and maintains webhook endpoint reliability. Trade-off is eventual consistency in payment status updates, requiring careful handling of race conditions where customers might see temporary status mismatches.

Processing Option	Reliability	Latency	Scalability	Complexity
Synchronous Processing	Medium (no retry)	Low	Limited	Low
Async Queue (Chosen)	High (built-in retry)	Medium	High	Medium
Hybrid Approach	High	Variable	Medium	High

The asynchronous queue approach provides the best balance of reliability and scalability. Critical payment status updates can still be handled synchronously during payment initiation, while webhook processing handles the authoritative state reconciliation asynchronously.

Webhook Idempotency and Deduplication

Payment gateways may send duplicate webhook events due to network issues, timeouts, or internal retry logic. The webhook processing system implements **idempotency protection** to ensure each event is processed exactly once:

- 1. Idempotency Key Extraction:** Extract the gateway-provided idempotency key from the webhook payload header
- 2. Duplicate Detection:** Check if this idempotency key has been processed before by querying the webhook event log
- 3. Atomic Processing:** If not a duplicate, process the webhook and record the idempotency key in a single database transaction
- 4. Idempotent Response:** Return success immediately for duplicate webhooks without reprocessing
- 5. Event Ordering:** Handle out-of-order webhooks by comparing event timestamps and only applying newer updates

The idempotency system maintains a webhook event log that serves both as duplicate protection and as an audit trail for debugging payment issues.

Payment Failure Recovery

When payment charges fail, the system implements a **graduated dunning process** that balances customer retention with payment collection. The recovery process includes multiple retry attempts with different strategies:

Retry Stage	Timing	Action Taken	Success Rate
Immediate Retry	1 hour after failure	Same payment method	~15% recovery
Backup Method	24 hours	Try backup payment method	~25% recovery
Customer Notification	3 days	Email with payment update link	~35% recovery
Account Suspension Warning	7 days	Final notice before suspension	~20% recovery
Service Suspension	14 days	Restrict account access	Payment required for restoration

Each retry attempt generates a new idempotency key to prevent conflicts with previous attempts. The system tracks retry history and automatically escalates through the dunning stages based on configurable business rules.

Implementation Guidance

The invoice generation and payment processing components require careful attention to financial accuracy, error handling, and integration patterns. The implementation balances real-time responsiveness with reliable background processing.

Technology Recommendations

| Component | Simple Option | Advanced Option | |---|---|---|---| | Invoice Generation | SQLite with JSON fields | PostgreSQL with JSONB and triggers | | Payment Gateway SDK | Stripe Python SDK | Multi-gateway abstraction layer | | Webhook Processing | Flask + Celery | FastAPI + Apache Kafka | | Currency Handling | Python Decimal with custom Money class | Babel + currency exchange APIs | | PDF Generation | ReportLab Python library | Headless Chrome with HTML templates | | Audit Logging | Structured JSON logging | Dedicated audit database |

The simple options provide rapid development velocity for proof-of-concept implementations, while advanced options support production-scale requirements with better performance and observability.

Recommended File Structure

```
billing_system/
  invoicing/
    __init__.py
    invoice_generator.py      ← Core invoice generation logic
    line_item_calculator.py  ← Individual charge calculations
    invoice_models.py        ← Invoice and LineItem data structures
    invoice_repository.py    ← Database operations
    invoice_service.py       ← High-level orchestration

  payments/
    __init__.py
    payment_gateway.py       ← Gateway abstraction interface
    stripe_gateway.py        ← Stripe-specific implementation
    webhook_handler.py      ← Webhook processing logic
    payment_models.py        ← Payment and Transaction structures
    payment_service.py       ← Payment orchestration

  shared/
    money.py                ← Currency and precision utilities
    audit_logger.py          ← Financial audit trail
    idempotency.py           ← Duplicate request protection

  tests/
    test_invoice_generation.py
    test_payment_processing.py
    test_webhook_handling.py
```

This structure separates invoice generation concerns from payment processing while providing shared utilities for financial operations. The clear separation enables independent testing and deployment of billing versus payment functionality.

Core Invoice Generation Infrastructure

The invoice generation system requires robust infrastructure for financial calculations and audit trails:

```
from decimal import Decimal, ROUND_HALF_UP

from dataclasses import dataclass, field

from typing import List, Dict, Optional

from datetime import datetime, timezone

from enum import Enum

import uuid

@dataclass

class Money:

    amount_cents: int

    currency_code: str


    @classmethod

    def from_decimal(cls, amount: Decimal, currency: str) -> 'Money':

        # Convert decimal to cents with proper rounding

        cents = int(amount.quantize(Decimal('0.01'), rounding=ROUND_HALF_UP) * 100)

        return cls(amount_cents=cents, currency_code=currency)


    def to_decimal(self) -> Decimal:

        return Decimal(self.amount_cents) / Decimal('100')


    def __add__(self, other: 'Money') -> 'Money':

        if self.currency_code != other.currency_code:

            raise ValueError("Cannot add different currencies")

        return Money(self.amount_cents + other.amount_cents, self.currency_code)


    def __sub__(self, other: 'Money') -> 'Money':
```

```
if self.currency_code != other.currency_code:

    raise ValueError("Cannot subtract different currencies")

return Money(self.amount_cents - other.amount_cents, self.currency_code)

class InvoiceStatus(Enum):

    DRAFT = "draft"

    OPEN = "open"

    PAID = "paid"

    PAST_DUE = "past_due"

    VOID = "void"

    UNCOLLECTIBLE = "uncollectible"

class LineItemType(Enum):

    SUBSCRIPTION = "subscription"

    USAGE = "usage"

    PRORATION = "proration"

    CREDIT = "credit"

    TAX = "tax"

@dataclass

class LineItem:

    line_item_id: uuid.UUID

    item_type: LineItemType

    description: str

    quantity: Decimal

    unit_price: Money

    total_amount: Money

    tax_amount: Money
```

```
metadata: Dict = field(default_factory=dict)

created_at: datetime = field(default_factory=lambda: datetime.now(timezone.utc))

@dataclass

class Invoice:

    invoice_id: uuid.UUID

    customer_id: uuid.UUID

    subscription_id: Optional[uuid.UUID]

    status: InvoiceStatus

    line_items: List[LineItem]

    subtotal_amount: Money

    tax_amount: Money

    credit_applied: Money

    total_amount: Money

    billing_period_start: datetime

    billing_period_end: datetime

    due_date: datetime

    created_at: datetime = field(default_factory=lambda: datetime.now(timezone.utc))

    updated_at: datetime = field(default_factory=lambda: datetime.now(timezone.utc))

    metadata: Dict = field(default_factory=dict)

class AuditEventType(Enum):

    INVOICE_CREATED = "invoice_created"

    INVOICE_FINALIZED = "invoice_finalized"

    PAYMENT_INITIATED = "payment_initiated"

    PAYMENT_SUCCEEDED = "payment_succeeded"

    PAYMENT_FAILED = "payment_failed"
```

```
CREDIT_APPLIED = "credit_applied"
```

This infrastructure provides type-safe financial operations with proper decimal precision and comprehensive audit trails. The `Money` class ensures all calculations maintain cent-level accuracy.

Payment Gateway Integration Skeleton

The payment gateway integration requires careful abstraction to support multiple payment providers:

```
from abc import ABC, abstractmethod

from typing import Dict, Optional, List

from dataclasses import dataclass

from enum import Enum

import uuid


class PaymentStatus(Enum):

    PENDING = "pending"

    PROCESSING = "processing"

    SUCCEEDED = "succeeded"

    FAILED = "failed"

    REQUIRES_ACTION = "requires_action"


@dataclass

class PaymentMethod:

    payment_method_id: str

    customer_id: uuid.UUID

    payment_type: str

    last_four: str

    expiration_month: Optional[int]

    expiration_year: Optional[int]

    is_default: bool

    metadata: Dict = field(default_factory=dict)


@dataclass

class PaymentResult:

    payment_id: str

    status: PaymentStatus
```

```
amount: Money

payment_method_id: str

gateway_transaction_id: Optional[str]

failure_reason: Optional[str]

requires_action: Optional[Dict] = None

metadata: Dict = field(default_factory=dict)

class PaymentGatewayInterface(ABC):

    """Abstract interface for payment gateway implementations."""

    @abstractmethod

    def create_charge(self, amount: Money, payment_method_id: str,
                      idempotency_key: str, metadata: Dict) -> PaymentResult:

        """Initiate a payment charge."""

        # TODO 1: Validate payment amount is positive

        # TODO 2: Retrieve payment method details from gateway

        # TODO 3: Create charge request with gateway-specific parameters

        # TODO 4: Handle immediate gateway response (success/failure/processing)

        # TODO 5: Return standardized PaymentResult

        pass

    @abstractmethod

    def retrieve_charge(self, charge_id: str) -> PaymentResult:

        """Get current status of an existing charge."""

        # TODO 1: Make gateway API call to retrieve charge details

        # TODO 2: Map gateway response to standardized PaymentResult

        # TODO 3: Handle gateway errors and timeouts gracefully
```

```
pass

@abstractmethod

def process_webhook(self, webhook_payload: str, signature: str) -> Optional[Dict]:

    """Process incoming webhook from payment gateway."""

    # TODO 1: Verify webhook signature for security

    # TODO 2: Parse webhook payload and extract event type

    # TODO 3: Validate event structure and required fields

    # TODO 4: Return standardized event data or None if invalid

    pass


@abstractmethod

def get_customer_payment_methods(self, customer_id: uuid.UUID) -> List[PaymentMethod]:

    """Retrieve all payment methods for a customer."""

    # TODO 1: Look up gateway customer ID from local mapping

    # TODO 2: Fetch payment methods from gateway API

    # TODO 3: Convert gateway response to PaymentMethod objects

    pass


class StripeGateway(PaymentGatewayInterface):

    """Stripe-specific implementation of payment gateway interface."""

    def __init__(self, api_key: str, webhook_secret: str):

        self.api_key = api_key

        self.webhook_secret = webhook_secret

        # TODO: Initialize Stripe SDK with API key
```

```
def create_charge(self, amount: Money, payment_method_id: str,
                  idempotency_key: str, metadata: Dict) -> PaymentResult:

    # TODO 1: Convert Money amount to Stripe's expected integer cents

    # TODO 2: Build Stripe PaymentIntent creation request

    # TODO 3: Include idempotency key in request headers

    # TODO 4: Handle Stripe-specific response format

    # TODO 5: Map Stripe status to PaymentStatus enum

    # Hint: Use stripe.PaymentIntent.create() with confirm=True for immediate
processing

    pass
```

The gateway abstraction enables testing with mock implementations and simplifies switching between payment providers. Each concrete implementation handles provider-specific API details while maintaining consistent interfaces.

Webhook Processing Service

Reliable webhook processing requires careful handling of authentication, deduplication, and failure recovery:

PYTHON

```
import json

import hmac

import hashlib

from typing import Optional, Dict, Any

from datetime import datetime, timezone

from dataclasses import dataclass

import uuid


@dataclass

class WebhookEvent:

    event_id: uuid.UUID

    gateway_event_id: str

    event_type: str

    processed_at: Optional[datetime]

    idempotency_key: str

    payload: Dict[str, Any]

    processing_attempts: int = 0

    created_at: datetime = field(default_factory=lambda: datetime.now(timezone.utc))



class WebhookProcessor:

    """Handles payment gateway webhook processing with idempotency and retry logic."""

    def __init__(self, payment_gateway: PaymentGatewayInterface,
                 database_manager: DatabaseManager):

        self.payment_gateway = payment_gateway

        self.db = database_manager


    def process_webhook(self, raw_payload: str, signature: str) -> Dict[str, Any]:
```

```
"""Process incoming webhook with full error handling and deduplication."""

# TODO 1: Verify webhook signature using payment gateway

# TODO 2: Parse payload and extract gateway event ID for idempotency

# TODO 3: Check if this webhook has been processed before

# TODO 4: If duplicate, return success without processing

# TODO 5: Process webhook event and update relevant entities

# TODO 6: Record successful processing in webhook event log

# TODO 7: Handle processing failures with appropriate error responses

# Hint: Use payment_gateway.process_webhook() for signature verification

pass

def handle_payment_succeeded(self, event_data: Dict[str, Any]) -> None:

    """Handle successful payment webhook events."""

    # TODO 1: Extract payment intent ID and amount from event data

    # TODO 2: Find corresponding invoice by payment intent reference

    # TODO 3: Update invoice status to PAID in database transaction

    # TODO 4: Update subscription status to active if payment resolves past due

    # TODO 5: Apply any remaining credit balance to customer account

    # TODO 6: Log audit event for payment completion

    # TODO 7: Trigger customer notification for successful payment

    pass

def handle_payment_failed(self, event_data: Dict[str, Any]) -> None:

    """Handle failed payment webhook events."""

    # TODO 1: Extract payment failure details and error codes

    # TODO 2: Find corresponding invoice and update status to PAST_DUE

    # TODO 3: Determine if failure is retryable or permanent
```

```

# TODO 4: Schedule dunning retry if appropriate

# TODO 5: Update subscription status based on dunning policy

# TODO 6: Log failure reason and next retry time

# TODO 7: Notify customer of payment failure with resolution options

pass

def verify_webhook_signature(self, payload: str, signature: str) -> bool:
    """Verify webhook authenticity using HMAC signature."""

    # TODO 1: Extract timestamp and signature from header

    # TODO 2: Build expected signature using webhook secret and payload

    # TODO 3: Compare signatures using constant-time comparison

    # TODO 4: Check timestamp to prevent replay attacks

    # TODO 5: Return True only if signature and timestamp are valid

    # Hint: Use hmac.compare_digest() for constant-time comparison

    pass

```

The webhook processor handles the complex orchestration between payment events and billing system state. Proper idempotency protection prevents duplicate processing even under high webhook volumes.

Milestone Checkpoints

After implementing the invoicing and payment processing components, verify functionality with these checkpoints:

Invoice Generation Verification:

1. Create a subscription with usage events and run invoice generation
2. Verify invoice contains correct line items for base subscription, usage overages, and credits
3. Check that all monetary calculations use integer cents and sum correctly
4. Confirm invoice status transitions from `draft` to `open` after finalization
5. Test proration calculations by changing plans mid-cycle

Payment Processing Verification:

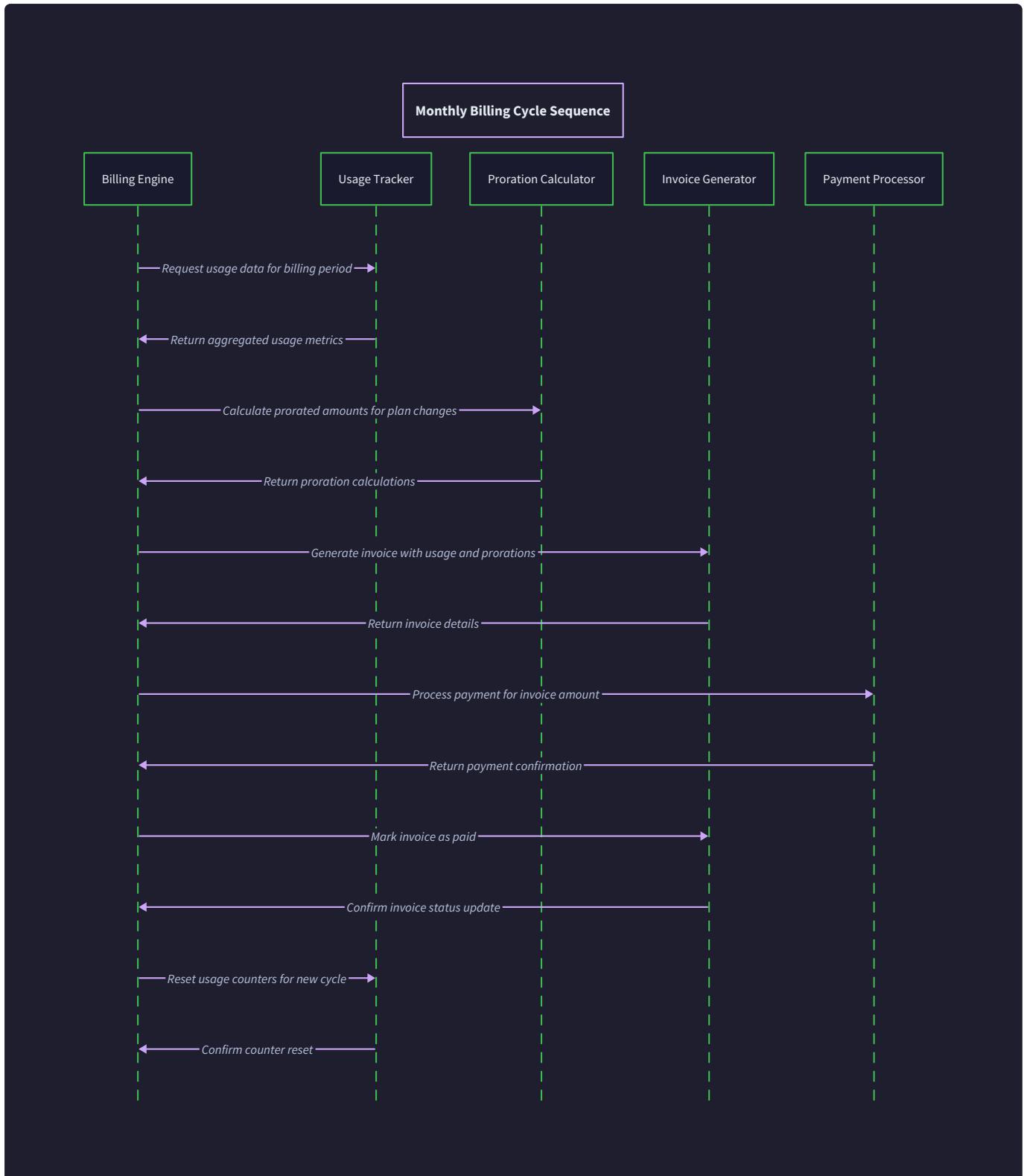
1. Submit a charge request and verify immediate response handling

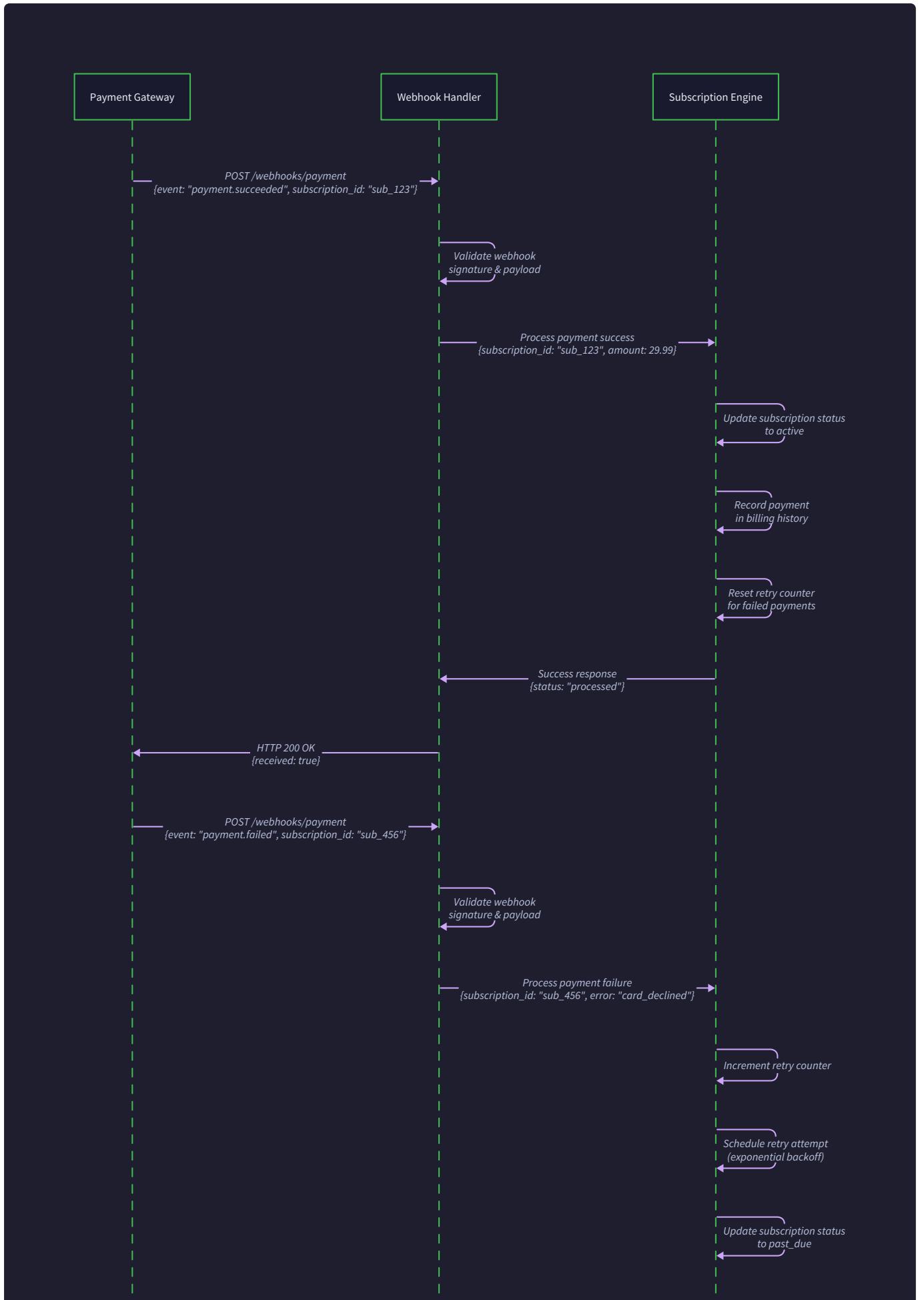
2. Simulate webhook events using payment gateway's testing tools
3. Confirm webhook deduplication by sending the same event twice
4. Test payment failure handling and dunning process initiation
5. Verify subscription status updates correctly based on payment outcomes

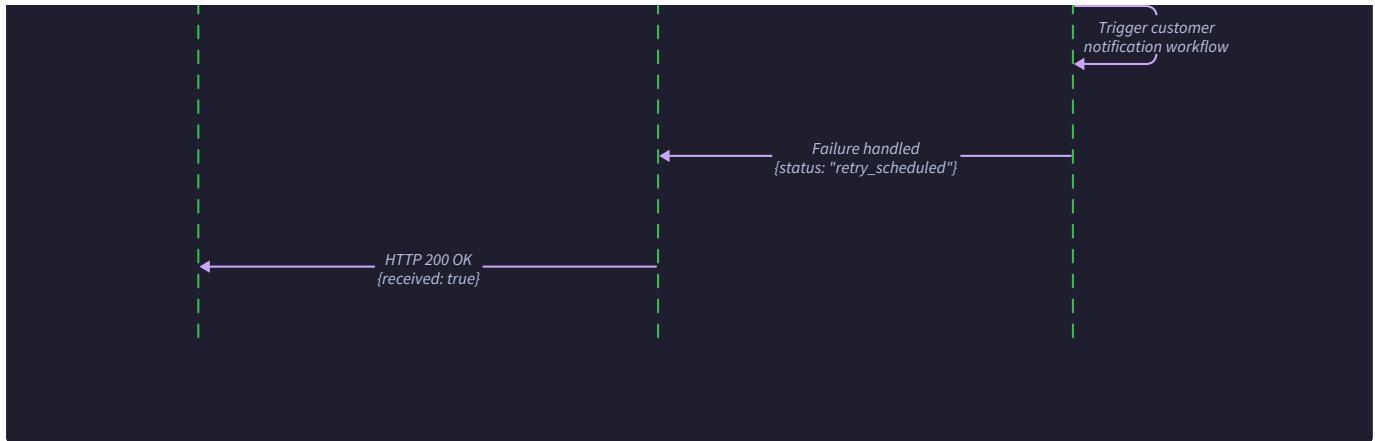
Integration Testing:

1. Run complete billing cycle from subscription renewal to payment completion
2. Test webhook processing under simulated network delays and failures
3. Verify audit trail captures all financial operations with proper attribution
4. Confirm currency precision maintained throughout all calculation steps

Expected behavior: Invoices should generate deterministically with identical line items for the same input data. Payment processing should handle both synchronous and asynchronous flows gracefully. All webhook events should process exactly once even under adverse network conditions.







Component Interactions and Data Flow

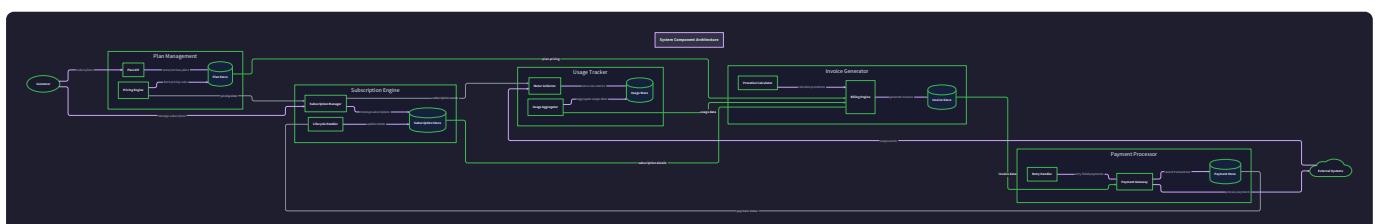
Milestone(s): All milestones - describes how components from plan management (Milestone 1), subscription lifecycle (Milestone 2), proration (Milestone 3), and usage-based billing (Milestone 4) work together

Mental Model: Orchestra Coordination

Think of the subscription billing system like a symphony orchestra performing a complex piece. Each component is a section of musicians (strings, brass, woodwinds, percussion) with their own specialized skills, but they must coordinate precisely to create beautiful music. The conductor (our event orchestration layer) ensures each section comes in at the right time with the right volume. When a customer upgrades their plan, it's like a musical transition where the strings fade out while the brass section crescendos - multiple components must work together seamlessly to create the desired effect without jarring the audience (our customers).

Just as musicians pass musical phrases between sections, our billing components pass data and events between each other. The subscription lifecycle component might start a "melody" by creating a new subscription, which the plan management component harmonizes by applying pricing rules, while the usage tracker adds percussion by measuring consumption, and finally the invoice generator brings it all together in the finale. When something goes wrong - like a musician missing their cue - the other sections must adapt gracefully to maintain the performance.

This mental model helps us understand that component interactions aren't just technical plumbing - they're choreographed workflows where timing, sequencing, and error handling are critical to delivering a smooth customer experience.



The subscription billing system orchestrates complex interactions between multiple specialized components. Each component has clear responsibilities, but they must coordinate precisely during key operations like subscription creation, monthly billing cycles, and plan changes. Understanding these interaction patterns is crucial for building a reliable billing system that maintains data consistency and provides a smooth customer experience.

Subscription Creation Flow

The subscription creation flow represents one of the most critical workflows in the billing system. It involves coordinating between plan management, payment processing, usage tracking initialization, and subscription state management. This flow must handle complex scenarios like trial periods, immediate charges, proration for mid-month starts, and graceful error recovery if any step fails.

End-to-End Process Flow

The subscription creation process follows a carefully orchestrated sequence that ensures data consistency and proper financial accounting. The flow begins when a customer selects a plan and provides payment information, and concludes with an active subscription that's ready for billing and usage tracking.

Step 1: Plan Validation and Feature Resolution The process starts with the Plan Management Component validating the selected plan and resolving all pricing and feature information. This includes checking plan availability, ensuring the plan supports the requested billing interval, and loading the complete feature entitlement matrix. The component also determines if the plan includes trial periods or requires immediate payment.

Step 2: Customer Credit Check and Payment Method Validation The system checks the customer's existing credit balance and validates the provided payment method through the Payment Gateway integration. This step ensures the customer has sufficient funds or valid payment instruments before proceeding with subscription creation. The payment method is tokenized for future recurring charges.

Step 3: Proration Calculation for Mid-Cycle Start If the subscription starts mid-cycle (not on the standard billing anchor day), the Proration Calculator determines the prorated charge for the partial billing period. This calculation considers the plan's billing interval, the current date, and the customer's preferred billing anchor day to ensure fair charging for partial periods.

Step 4: Initial Invoice Generation The Invoice Generator creates the first invoice for the subscription, including any trial period handling, prorated charges, setup fees, and applicable taxes. If the customer has existing credit balance, it's automatically applied to reduce the invoice amount. The invoice includes detailed line items showing exactly what the customer is being charged for.

Step 5: Payment Processing and Authorization The Payment Processor attempts to charge the customer's payment method for any immediate amounts due. For trial subscriptions, this might only authorize the payment method without charging. The system handles various payment scenarios including successful charges, failed payments, and payment methods requiring additional authentication.

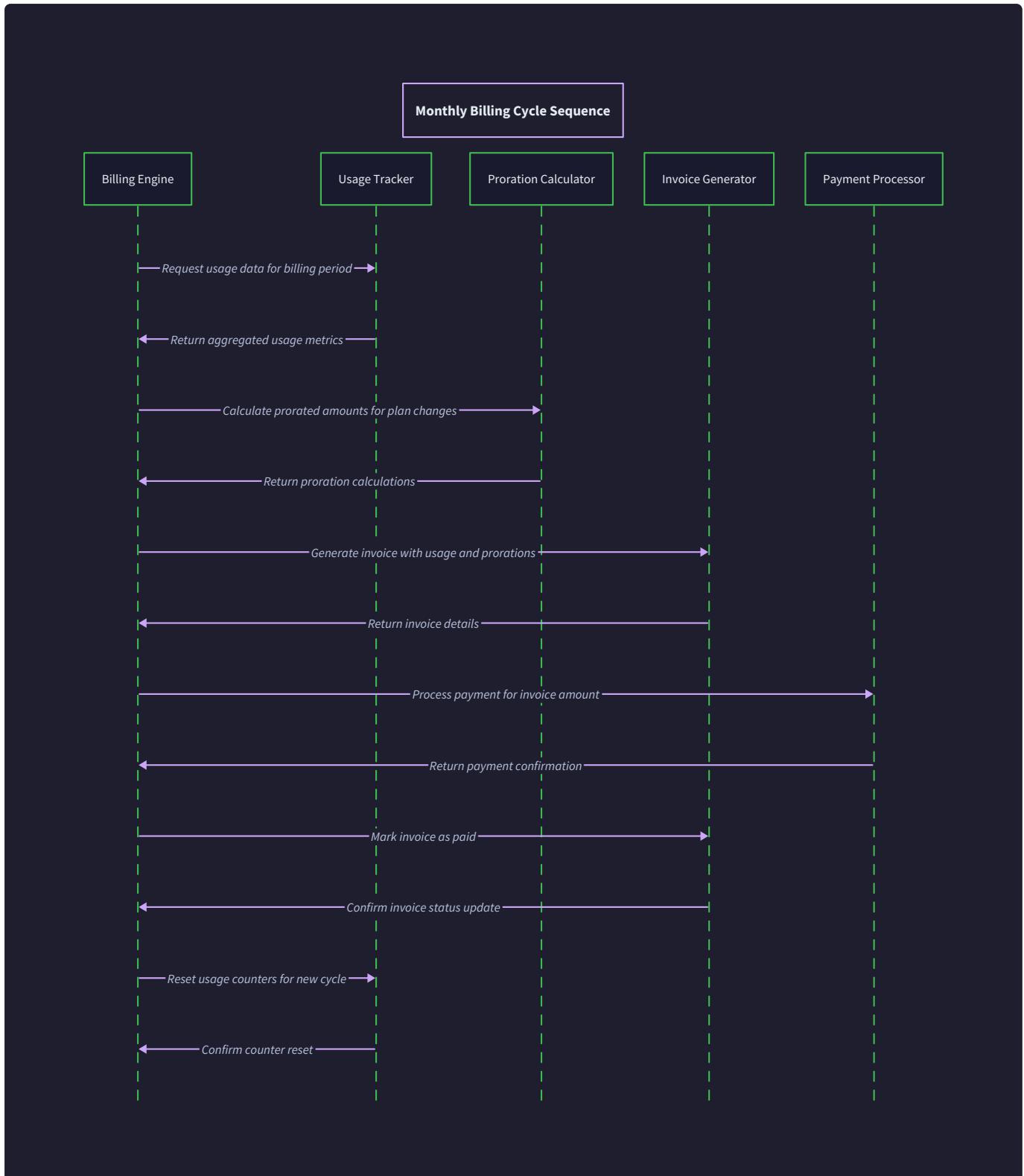
Step 6: Subscription Record Creation and State Initialization Upon successful payment (or trial authorization), the Subscription Engine creates the subscription record with the appropriate initial state. Trial subscriptions start in `trialing` status, while paid subscriptions begin in `active` status. The subscription includes billing anchor calculation, next billing date determination, and feature entitlement activation.

Step 7: Usage Tracking Initialization The Usage-Based Billing Engine initializes usage tracking for the new subscription. This includes setting up usage allowances based on the plan, creating initial usage aggregation records, and configuring any quota limits. The system prepares to track metered usage from the subscription start date.

Step 8: Welcome Communications and Service Provisioning Finally, the system triggers welcome communications to the customer and provisions access to subscribed services. This includes sending confirmation emails, updating customer portal access, and notifying downstream services about the new subscription and its feature entitlements.

Component Interaction Patterns

The subscription creation flow demonstrates several important interaction patterns that appear throughout the billing system. Understanding these patterns helps developers build consistent and reliable workflows.



Interaction Pattern	Description	Components Involved	Data Exchanged
Synchronous Validation	Immediate validation with error response	Plan Management, Payment Gateway	Plan details, payment method validation
Asynchronous Processing	Background tasks triggered by events	Invoice Generator, Email Service	Subscription events, notification requests
Transactional Coordination	Multi-step operations with rollback capability	Database Manager, All Components	Entity creation, state changes
Event-Driven Updates	Components react to subscription lifecycle events	Event Bus, Usage Tracker, Audit Logger	Subscription events, audit records
Credit Application	Automatic credit consumption during billing	Credit Balance Manager, Invoice Generator	Credit amounts, invoice adjustments

Error Handling and Rollback Scenarios

The subscription creation flow must handle various failure modes gracefully. Each step in the process includes specific error handling and rollback procedures to ensure the system remains in a consistent state.

If payment processing fails after the subscription record is created, the system automatically transitions the subscription to `incomplete` status and schedules payment retry attempts. The customer retains access to trial features but cannot access paid features until payment succeeds.

When plan validation fails due to plan unavailability or pricing errors, the system returns detailed error messages to help customers select alternative plans. No subscription record is created, and no charges are attempted.

Database transaction failures during subscription creation trigger automatic rollback of all related records, including invoice line items, usage tracking setup, and payment authorizations. The system logs detailed error information for debugging while providing user-friendly error messages to customers.

Data Consistency Guarantees

The subscription creation flow maintains strict data consistency through several mechanisms. All financial operations occur within database transactions with serializable isolation levels to prevent race conditions. The system uses idempotency keys to ensure duplicate subscription creation requests don't result in multiple charges or subscriptions.

Event publishing follows the transactional outbox pattern, where events are written to the database as part of the same transaction that creates the subscription, then published asynchronously. This ensures that downstream systems receive notifications about all successfully created subscriptions without missing events due to system failures.

The flow also implements compensation patterns for distributed operations. If payment succeeds but subscription creation fails, the system automatically initiates refund processing to prevent charging customers

for non-existent subscriptions.

Monthly Billing Cycle Flow

The monthly billing cycle represents the heartbeat of the subscription system. This process runs automatically for all active subscriptions, calculating charges, applying usage-based billing, processing payments, and handling failures. The billing cycle must process potentially thousands of subscriptions reliably while maintaining strict financial accuracy and providing detailed audit trails.

Billing Cycle Orchestration

The billing cycle operates as a distributed workflow that processes subscriptions in carefully coordinated stages. The system uses a combination of scheduled triggers and event-driven processing to ensure reliable execution even during high-volume periods or system failures.

Stage 1: Billing Period Identification and Subscription Selection The billing process begins by identifying all subscriptions due for renewal on the current date. The system queries subscriptions where the next billing date matches the current date and the subscription status allows billing (active, trialing with trial ending). This selection process considers timezone differences to ensure customers are billed at the appropriate time in their local timezone.

The system also identifies subscriptions transitioning from trial to paid status on the current date. These subscriptions require special handling to activate paid features and process the first recurring charge.

Stage 2: Usage Aggregation and Overage Calculation For each subscription due for billing, the Usage-Based Billing Engine aggregates all usage events from the previous billing period. This aggregation process totals consumption by usage type and compares against plan allowances to calculate any overage charges.

The aggregation uses precise billing period boundaries, starting from the last billing date (inclusive) to the current billing date (exclusive). This ensures usage is attributed to the correct billing period even when usage events arrive with slight timestamp delays.

Stage 3: Invoice Generation with Line Item Breakdown The Invoice Generator creates detailed invoices for each subscription, including multiple line item types. Base subscription charges appear as recurring line items, while usage overages create separate metered line items with quantity and rate breakdowns.

The invoice generation process automatically applies any existing customer credit balance, reducing the total amount due. Credits are consumed on a first-in-first-out basis, with detailed line items showing credit application amounts and remaining balances.

Stage 4: Payment Processing and Retry Logic The Payment Processor attempts to charge each customer's default payment method for the invoice total. The system implements intelligent retry logic for failed payments, using exponential backoff and multiple payment method fallback.

Successful payments immediately mark invoices as paid and maintain subscription active status. Failed payments trigger the dunning management process, transitioning subscriptions to `past_due` status and scheduling retry attempts.

Stage 5: Subscription Renewal and Next Billing Date Calculation Successfully billed subscriptions have their subscription records updated with new billing period information. The system calculates the next billing date based on the subscription's billing interval and billing anchor day, handling edge cases like month-end dates and leap years.

The subscription's feature entitlements are refreshed based on the current plan version, and usage allowances are reset for the new billing period. Any plan changes scheduled for the renewal date are applied at this stage.

Stage 6: Usage Tracking Reset and Quota Reactivation The Usage-Based Billing Engine resets usage tracking for the new billing period. Previous period usage is archived for reporting purposes, while current period counters are zeroed out. Any quota restrictions imposed due to overage in the previous period are lifted if payment succeeded.

The system also recalculates usage limits based on any plan changes that took effect during renewal, ensuring customers have access to their updated usage allowances immediately.

Failure Handling and Recovery

The monthly billing cycle implements comprehensive failure handling to ensure financial accuracy and customer satisfaction. Each stage of the billing process includes specific failure modes and recovery procedures.

Payment Failure Recovery When payment processing fails, the system distinguishes between temporary failures (insufficient funds, network issues) and permanent failures (expired cards, closed accounts).

Temporary failures trigger automatic retry attempts with increasing delays, while permanent failures require customer intervention to update payment methods.

The dunning management system sends progressively more urgent communications to customers about failed payments, while maintaining service access for a grace period. Customers retain access to their subscribed services but may have certain features restricted until payment succeeds.

Usage Aggregation Failures If usage aggregation fails for a subscription, the billing cycle continues with base subscription charges while marking the usage calculation for manual review. This ensures customers aren't blocked from service due to usage tracking issues, while finance teams can resolve discrepancies later.

The system maintains detailed logs of aggregation failures, including which usage events couldn't be processed and why. This information enables rapid resolution of billing disputes and usage tracking bugs.

Database Consistency During High Volume During peak billing periods, the system uses optimistic locking and retry mechanisms to handle database contention. Each subscription's billing operation occurs within its own transaction to prevent failures in one subscription from affecting others.

The billing process can be safely restarted at any stage, using idempotency keys and billing status flags to resume from the last successful checkpoint. This allows the system to recover quickly from infrastructure failures during critical billing windows.

Performance and Scalability Considerations

The monthly billing cycle must handle varying loads efficiently, from systems with hundreds of subscriptions to enterprise platforms processing millions of renewals. The system uses several techniques to maintain performance at scale.

Batch Processing with Parallel Execution Subscriptions are processed in configurable batch sizes, with multiple batches running in parallel. The batch size balances memory usage against processing efficiency, while parallel execution utilizes available system resources effectively.

Each batch processes subscriptions independently, allowing the system to scale horizontally by adding more worker processes or servers. Failed batches can be retried independently without affecting successful processing.

Database Query Optimization The billing process uses optimized database queries with appropriate indexes on billing-related fields. Usage aggregation queries use time-range partitioning to scan only relevant data, while subscription selection queries use compound indexes on status and next billing date.

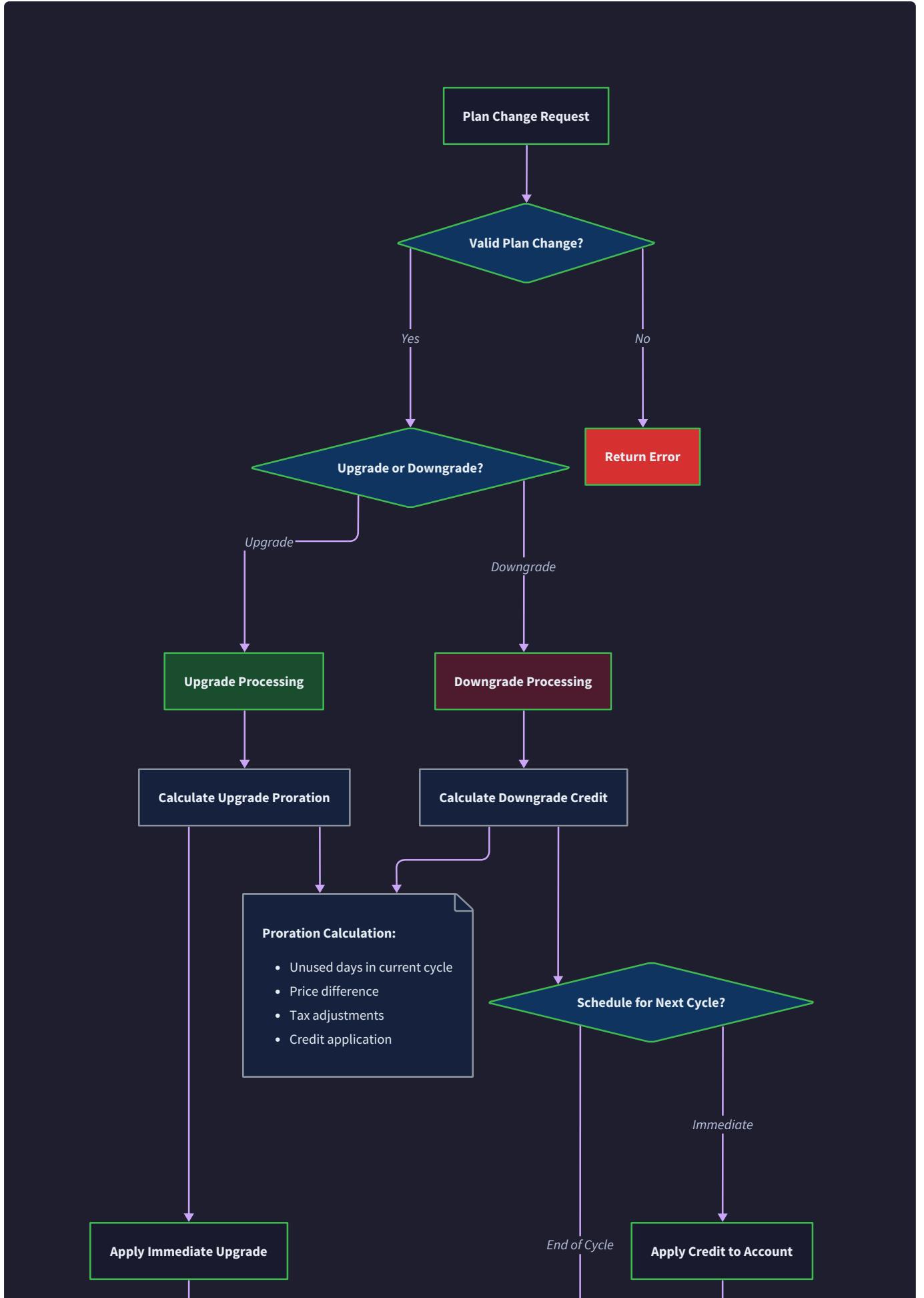
The system pre-loads related data (plans, payment methods, customer information) in batch queries to minimize database round-trips during processing. This approach significantly reduces the per-subscription processing time.

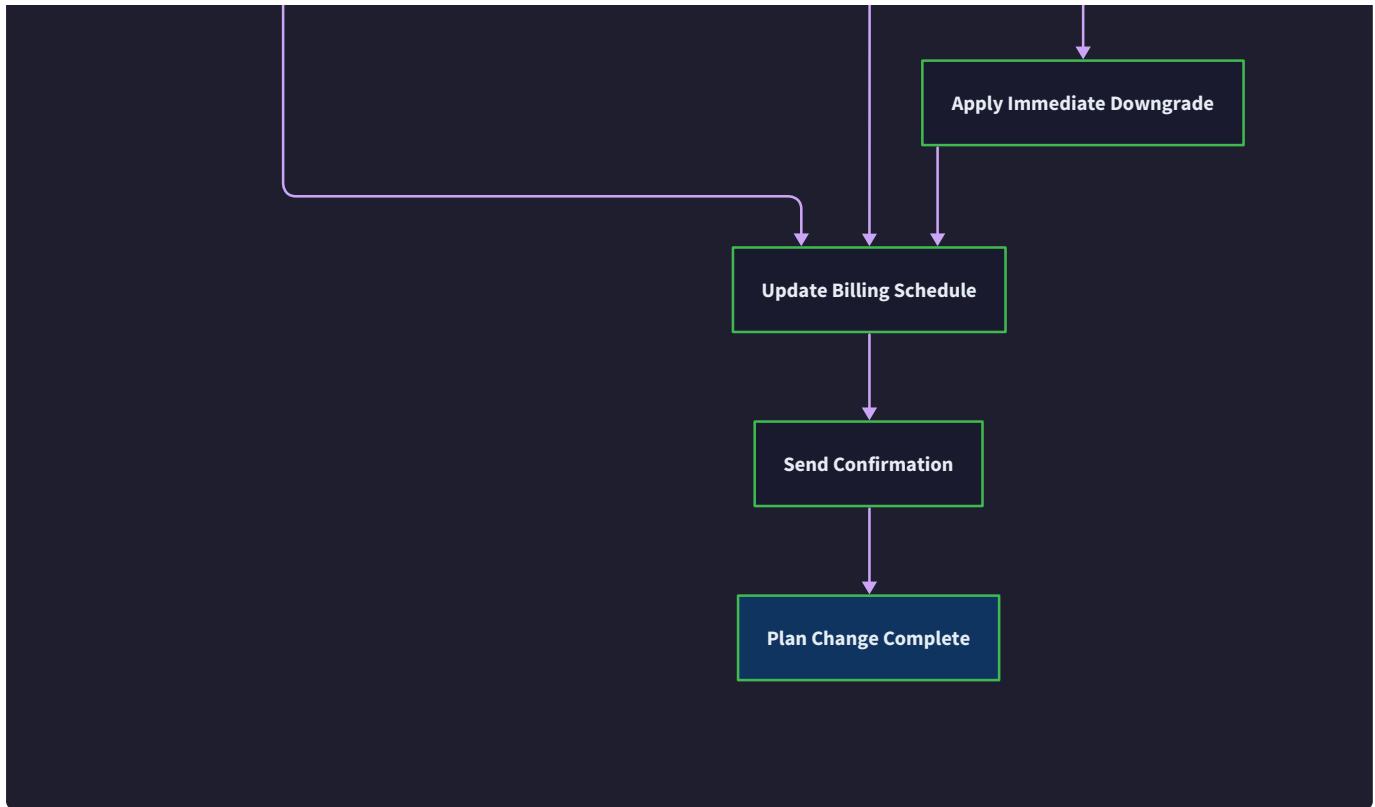
Memory Management and Resource Cleanup Large-scale billing operations carefully manage memory usage to prevent out-of-memory errors during peak periods. Usage aggregation results are processed in streaming fashion rather than loading entire datasets into memory.

The system includes monitoring and alerting for billing cycle performance, tracking metrics like processing time per subscription, memory usage, and error rates. This visibility enables proactive capacity planning and performance optimization.

Plan Change and Proration Flow

Plan changes represent some of the most complex operations in subscription billing, requiring careful coordination between multiple components to ensure accurate financial calculations and smooth customer experience. The process must handle upgrades, downgrades, immediate changes, scheduled changes, and various edge cases while maintaining strict financial accuracy.





Plan Change Orchestration Workflow

The plan change process follows a sophisticated workflow that balances customer expectations with financial precision. Customers expect plan changes to take effect immediately or at predictable times, while the billing system must ensure accurate proration and proper credit handling.

Phase 1: Plan Change Validation and Authorization The process begins with comprehensive validation of the requested plan change. The Plan Management Component verifies that the target plan is available, supports the customer's billing preferences, and allows the specific transition from the current plan.

The system checks whether the change represents an upgrade (higher cost) or downgrade (lower cost) to determine the appropriate proration and payment handling. Upgrade validations include verifying the customer's payment method can handle additional charges, while downgrades require checking refund policies and credit balance limits.

Authorization checks ensure the customer has permission to make plan changes and hasn't exceeded any change frequency limits. Some businesses restrict plan changes to prevent abuse or reduce support complexity.

Phase 2: Proration Calculation and Financial Impact Analysis The Proration Calculator performs detailed financial analysis of the plan change, determining exactly how much the customer should be charged or credited. This calculation considers the time remaining in the current billing period, the difference between old and new plan pricing, and any usage-based charges that need adjustment.

For upgrades, the system calculates the prorated charge for the remaining days in the billing cycle at the new plan rate, minus the value of unused time at the old rate. For downgrades, it calculates credits for the unused portion of the higher-priced plan.

The calculation also considers usage allowances and overages. If the new plan has different usage limits, the system recalculates overage charges based on current period usage and the new plan's allowances.

Phase 3: Invoice Adjustment and Credit Processing Based on the proration calculation, the Invoice Generator creates adjustment line items for the customer's next invoice or generates an immediate invoice for upgrades requiring payment. Credit adjustments are processed through the Credit Balance Manager to ensure proper accounting and audit trails.

Upgrades typically result in immediate charges for the prorated difference, while downgrades create credits applied to future invoices. The system generates detailed line items explaining each adjustment to provide transparency to customers and support teams.

Phase 4: Subscription Record Updates and Feature Activation The Subscription Engine updates the subscription record with the new plan information, adjusting feature entitlements immediately. Customers gain access to new features right away, even if the financial adjustments are processed asynchronously.

The subscription's next billing date remains unchanged to maintain the existing billing cycle anchor. However, the system records the plan change effective date for accurate future billing calculations.

Phase 5: Usage Tracking Adjustment and Quota Updates The Usage-Based Billing Engine adjusts usage tracking for the new plan's allowances and quotas. If the new plan includes higher usage limits, those become available immediately. If the new plan has lower limits and current usage exceeds them, the system applies appropriate overage charges or quota restrictions.

Usage tracking maintains historical records of plan changes to ensure accurate aggregation and billing at the end of the billing period. Each usage event is associated with the plan that was active when the usage occurred.

Phase 6: Downstream Service Notifications and Access Updates Finally, the system notifies all downstream services about the plan change and updated feature entitlements. This includes updating customer portal access, API rate limits, storage quotas, and any other plan-dependent service configurations.

These notifications use the event-driven architecture to ensure all services receive updates without requiring direct coupling between the billing system and service implementations.

Complex Proration Scenarios

The plan change flow must handle numerous complex scenarios that arise in real-world billing operations. Each scenario requires specific handling to ensure fairness and accuracy.

Multiple Plan Changes Within a Billing Period When customers make multiple plan changes within a single billing period, the system maintains a chronological record of all changes and calculates cumulative proration effects. Each change is treated as a separate proration calculation, with the effective date determining how much of the billing period applies to each plan.

The system prevents proration gaming by tracking change frequency and applying business rules about minimum time between changes or maximum changes per period. This protects against customers attempting

to exploit proration calculations.

Plan Changes with Different Billing Intervals Changing from monthly to annual billing (or vice versa) requires special handling since the billing cycles don't align. The system calculates proration for the remainder of the current billing cycle, then schedules the subscription for renewal on the new billing interval.

These changes often involve significant proration amounts, either charging for several months upfront (monthly to annual) or crediting unused annual subscription time (annual to monthly). The system provides clear explanations of these calculations to customers.

Usage-Based Plan Changes with Existing Consumption When changing plans mid-cycle with existing usage consumption, the system must recalculate overage charges based on each plan's allowances and rates. Usage that was within allowances under the old plan might become overage under a new plan with lower limits.

The system applies a fairness algorithm that ensures customers aren't double-charged for usage but also aren't given unfair advantages by switching plans to avoid overage charges they've already incurred.

Error Handling and Rollback Mechanisms

Plan changes involve multiple financial and operational systems, creating numerous potential failure points. The system implements comprehensive error handling and rollback capabilities to maintain consistency.

Financial Rollback for Payment Failures If a plan upgrade requires immediate payment and the charge fails, the system automatically rolls back the plan change, restoring the previous plan and feature entitlements. Any credits or adjustments created during the change process are reversed.

The rollback process maintains audit trails of both the attempted change and the rollback, ensuring complete transparency for customer service and accounting teams.

Partial Failure Recovery When some aspects of a plan change succeed (like updating the subscription record) but others fail (like processing payments or updating downstream services), the system uses compensation patterns to restore consistency.

The system maintains a plan change state machine that tracks the progress of each change operation, enabling automatic retry of failed steps or manual intervention when necessary.

Data Consistency During Concurrent Operations Plan changes use optimistic locking to prevent conflicts when multiple operations target the same subscription simultaneously. If a plan change conflicts with billing cycle processing or other operations, the system queues the change for execution after the conflicting operation completes.

This approach ensures that plan changes don't interfere with critical billing operations while still providing responsive plan change processing for customers.

Key Design Insight: The component interaction patterns in subscription billing mirror those found in distributed financial systems - they require careful coordination, comprehensive error handling, and strong consistency guarantees. The complexity arises not from individual operations but from the interactions between components that must maintain financial accuracy while providing responsive customer experiences.

Implementation Guidance

The component interaction flows represent the most complex aspects of subscription billing implementation. These workflows coordinate multiple specialized components and require careful attention to error handling, data consistency, and performance. The following guidance provides concrete starting points for implementing these critical system workflows.

Technology Recommendations

Component	Simple Option	Advanced Option
Workflow Orchestration	Direct function calls with try/catch	Temporal.io or Conductor workflow engine
Event Publishing	In-memory event bus with immediate processing	Apache Kafka with persistent message queues
Database Transactions	Single database with ACID transactions	Distributed transactions with saga pattern
Error Recovery	Manual retry with exponential backoff	Circuit breaker pattern with automatic failover
Monitoring	Structured logging with correlation IDs	Distributed tracing with OpenTelemetry
State Management	Database-backed state machines	Redis-based distributed state with TTL

Recommended File Structure

```
subscription-billing/
├── internal/
│   ├── workflows/           ← component interaction orchestration
│   │   ├── subscription_creation.py ← subscription creation workflow
│   │   ├── billing_cycle.py      ← monthly billing cycle workflow
│   │   ├── plan_change.py      ← plan change and proration workflow
│   │   └── workflow_base.py    ← common workflow infrastructure
│   ├── events/              ← event-driven communication
│   │   ├── event_bus.py        ← event publishing and subscription
│   │   ├── event_types.py     ← event type definitions
│   │   └── handlers/          ← event handler implementations
│   │       ├── audit_handler.py ← audit trail event handler
│   │       ├── notification_handler.py ← customer notification handler
│   │       └── analytics_handler.py ← usage analytics handler
│   ├── coordinators/        ← workflow coordination logic
│   │   ├── billing_coordinator.py ← coordinates billing cycle operations
│   │   ├── payment_coordinator.py ← coordinates payment processing
│   │   └── proration_coordinator.py ← coordinates proration calculations
│   └── state_machines/       ← subscription state management
│       ├── subscription_state.py ← subscription lifecycle state machine
│       ├── invoice_state.py    ← invoice processing state machine
│       └── payment_state.py   ← payment processing state machine
└── tests/
    ├── integration/          ← end-to-end workflow tests
    │   ├── test_subscription_creation_flow.py
    │   ├── test_billing_cycle_flow.py
    │   └── test_plan_change_flow.py
    └── workflows/             ← workflow unit tests
        ├── run_billing_cycle.py   ← manual billing cycle trigger
        └── fix_failed_subscriptions.py ← recovery script for failed operations
```

Workflow Infrastructure Starter Code

```
# internal/workflows/workflow_base.py                                PYTHON

from abc import ABC, abstractmethod

from dataclasses import dataclass

from typing import Dict, Any, Optional, List

from enum import Enum

import uuid

from datetime import datetime

import logging

logger = logging.getLogger(__name__)

class WorkflowStatus(Enum):

    PENDING = "pending"

    RUNNING = "running"

    COMPLETED = "completed"

    FAILED = "failed"

    COMPENSATING = "compensating" # rolling back due to failure

    @dataclass

    class WorkflowStep:

        step_id: str

        name: str

        status: WorkflowStatus

        input_data: Dict[str, Any]

        output_data: Optional[Dict[str, Any]] = None

        error_message: Optional[str] = None

        started_at: Optional[datetime] = None
```

```
completed_at: Optional[datetime] = None

retry_count: int = 0


@dataclass

class WorkflowExecution:

    execution_id: str

    workflow_type: str

    status: WorkflowStatus

    input_data: Dict[str, Any]

    steps: List[WorkflowStep]

    created_at: datetime

    updated_at: datetime

    correlation_id: Optional[str] = None


class WorkflowExecutor(ABC):

    """Base class for all workflow executors that coordinate component interactions."""

    def __init__(self, db_manager: DatabaseManager, event_bus: EventBus):

        self.db_manager = db_manager

        self.event_bus = event_bus

        self.max_retries = 3


    @abstractmethod

    def get_workflow_steps(self) -> List[str]:

        """Return ordered list of step names for this workflow."""

        pass


    @abstractmethod
```

```
async def execute_step(self, step_name: str, step_input: Dict[str, Any],  
                      execution: WorkflowExecution) -> Dict[str, Any]:  
  
    """Execute a single workflow step and return output data."""  
  
    pass  
  
  
    @abstractmethod  
  
    async def compensate_step(self, step_name: str, step_output: Dict[str, Any],  
                            execution: WorkflowExecution) -> None:  
  
        """Rollback/compensate for a completed step during failure recovery."""  
  
        pass  
  
  
    async def execute_workflow(self, input_data: Dict[str, Any],  
                             correlation_id: Optional[str] = None) -> WorkflowExecution:  
  
        """Execute complete workflow with error handling and compensation."""  
  
        execution_id = str(uuid.uuid4())  
  
        execution = WorkflowExecution(  
  
            execution_id=execution_id,  
  
            workflow_type=self.__class__.__name__,  
  
            status=WorkflowStatus.PENDING,  
  
            input_data=input_data,  
  
            steps=[],  
  
            created_at=datetime.utcnow(),  
  
            updated_at=datetime.utcnow(),  
  
            correlation_id=correlation_id  
  
)  
  
  
    # Initialize workflow steps
```

```
for step_name in self.get_workflow_steps():

    step = WorkflowStep(
        step_id=str(uuid.uuid4()),
        name=step_name,
        status=WorkflowStatus.PENDING,
        input_data={}
    )

    execution.steps.append(step)

# Persist initial workflow state

await self._save_execution(execution)

try:

    execution.status = WorkflowStatus.RUNNING

    await self._save_execution(execution)

# Execute each step in sequence

    step_input = input_data

    for step in execution.steps:

        step.input_data = step_input

        step.status = WorkflowStatus.RUNNING

        step.started_at = datetime.utcnow()

        await self._save_execution(execution)

    try:

        step_output = await self.execute_step(step.name, step_input, execution)

        step.output_data = step_output
```

```
        step.status = WorkflowStatus.COMPLETED

        step.completed_at = datetime.utcnow()

        step_input = step_output # Output becomes input for next step


    except Exception as e:

        step.status = WorkflowStatus.FAILED

        step.error_message = str(e)

        step.completed_at = datetime.utcnow()

        await self._save_execution(execution)

        # Trigger compensation workflow

        await self._compensate_workflow(execution)

        raise e

    await self._save_execution(execution)

execution.status = WorkflowStatus.COMPLETED

execution.updated_at = datetime.utcnow()

await self._save_execution(execution)

# Publish workflow completion event

await self.event_bus.publish(SubscriptionEvent(
    event_id=uuid.uuid4(),
    event_type=SubscriptionEventType.WORKFLOW_COMPLETED,
    subscription_id=input_data.get('subscription_id'),
    data={'execution_id': execution_id, 'workflow_type':
execution.workflow_type},
```

```
        idempotency_key=f"workflow_completed_{execution_id}",
        created_at=datetime.utcnow()
    ))

    return execution

except Exception as e:
    execution.status = WorkflowStatus.FAILED
    execution.updated_at = datetime.utcnow()
    await self._save_execution(execution)
    logger.error(f"Workflow {execution_id} failed: {e}")
    raise

async def _compensate_workflow(self, execution: WorkflowExecution) -> None:
    """Execute compensation steps in reverse order for failed workflow."""
    execution.status = WorkflowStatus.COMPENSATING
    await self._save_execution(execution)

    # Compensate completed steps in reverse order
    completed_steps = [s for s in reversed(execution.steps)
                       if s.status == WorkflowStatus.COMPLETED]

    for step in completed_steps:
        try:
            await self.compensate_step(step.name, step.output_data or {}, execution)
            logger.info(f"Compensated step {step.name} for workflow
{execution.execution_id}")
        except Exception as e:
            logger.error(f"Failed to compensate step {step.name} for workflow
{execution.execution_id}: {e}")
```

```
        except Exception as e:
            logger.error(f"Failed to compensate step {step.name}: {e}")

            # Continue compensation of other steps even if one fails


async def _save_execution(self, execution: WorkflowExecution) -> None:
    """Persist workflow execution state to database."""

    # TODO: Implement workflow execution persistence

    # This should save the complete execution state including all steps
    # to allow for workflow recovery and monitoring

    pass
```

Event-Driven Communication Infrastructure

```
# internal/events/event_bus.py                                         PYTHON

from typing import Dict, List, Callable, Any

from dataclasses import dataclass

from datetime import datetime

import uuid

import asyncio

import json

import logging

logger = logging.getLogger(__name__)

@dataclass

class Event:

    event_id: uuid.UUID

    event_type: str

    data: Dict[str, Any]

    timestamp: datetime

    correlation_id: Optional[str] = None

    idempotency_key: Optional[str] = None

class EventBus:

    """Event bus for component communication with guaranteed delivery and deduplication."""

    def __init__(self, db_manager: DatabaseManager):

        self.db_manager = db_manager

        self.handlers: Dict[str, List[Callable]] = {}

        self.processed_events: set = set() # For idempotency
```

```
def subscribe(self, event_type: str, handler: Callable[[Event], None]) -> None:
    """Subscribe handler function to specific event type."""

    if event_type not in self.handlers:

        self.handlers[event_type] = []

    self.handlers[event_type].append(handler)

async def publish(self, event: Event) -> bool:
    """Publish event to all registered handlers with idempotency protection."""

    # Check idempotency

    if event.idempotency_key and event.idempotency_key in self.processed_events:

        logger.info(f"Skipping duplicate event {event.idempotency_key}")

        return True

    try:

        # Persist event using transactional outbox pattern

        async with
self.db_manager.transaction(isolation_level=ISOLATION_LEVEL_SERIALIZABLE):

            await self._persist_event(event)

            # Mark as processed for idempotency

            if event.idempotency_key:

                self.processed_events.add(event.idempotency_key)

            # Publish to handlers asynchronously

            await self._notify_handlers(event)

    return True
```

```
except Exception as e:

    logger.error(f"Failed to publish event {event.event_id}: {e}")

    return False


async def _persist_event(self, event: Event) -> None:

    """Persist event to outbox table for guaranteed delivery."""

    # TODO: Insert event into outbox table

    # This ensures events are never lost even if handler processing fails

    pass


async def _notify_handlers(self, event: Event) -> None:

    """Notify all registered handlers for event type."""

    if event.event_type not in self.handlers:

        return

    tasks = []

    for handler in self.handlers[event.event_type]:

        task = asyncio.create_task(self._safe_handler_call(handler, event))

        tasks.append(task)

    if tasks:

        await asyncio.gather(*tasks, return_exceptions=True)


async def _safe_handler_call(self, handler: Callable, event: Event) -> None:

    """Call event handler with error isolation."""

    try:
```

```
if asyncio.iscoroutinefunction(handler):

    await handler(event)

else:

    handler(event)

except Exception as e:

    logger.error(f"Event handler {handler.__name__} failed for event
{event.event_id}: {e}")

    # Handler failures don't prevent other handlers from running
```

Subscription Creation Workflow Implementation

```
# internal/workflows/subscription_creation.py                                PYTHON

from internal.workflows.workflow_base import WorkflowExecutor, WorkflowStep

from internal.plan_management import PlanManager

from internal.subscription_engine import SubscriptionEngine

from internal.proration import ProrationCalculator

from internal.invoice_generator import InvoiceGenerator

from internal.payment_processor import PaymentProcessor

from internal.usage_tracking import UsageTracker


class SubscriptionCreationWorkflow(WorkflowExecutor):

    """Orchestrates subscription creation across multiple components."""

    def __init__(self, db_manager: DatabaseManager, event_bus: EventBus,
                 plan_manager: PlanManager, subscription_engine: SubscriptionEngine,
                 proration_calculator: ProrationCalculator, invoice_generator:
                 InvoiceGenerator,
                 payment_processor: PaymentProcessor, usage_tracker: UsageTracker):
        super().__init__(db_manager, event_bus)

        self.plan_manager = plan_manager

        self.subscription_engine = subscription_engine

        self.proration_calculator = proration_calculator

        self.invoice_generator = invoice_generator

        self.payment_processor = payment_processor

        self.usage_tracker = usage_tracker


    def get_workflow_steps(self) -> List[str]:
        return [
```

```
        "validate_plan",
        "check_customer_credit",
        "calculate_proration",
        "generate_invoice",
        "process_payment",
        "create_subscription",
        "initialize_usage_tracking"
    ]
}

async def execute_step(self, step_name: str, step_input: Dict[str, Any],
                      execution: WorkflowExecution) -> Dict[str, Any]:
    """Execute individual workflow step."""

    if step_name == "validate_plan":
        # TODO 1: Extract plan_id from step_input
        # TODO 2: Call self.plan_manager.get_plan(plan_id) to load plan details
        # TODO 3: Validate plan is available and supports requested billing_interval
        # TODO 4: Check if plan has trial_period_days > 0 for trial handling
        # TODO 5: Return dict with plan details, is_trial flag, and validation status
        pass

    elif step_name == "check_customer_credit":
        # TODO 1: Extract customer_id from step_input
        # TODO 2: Get customer credit balance using credit_balance_manager
        # TODO 3: Validate payment_method_id through payment processor
        # TODO 4: Check if customer has sufficient credit to cover setup fees
        # TODO 5: Return dict with credit_balance_cents and payment_method_valid flag
```

```
pass

elif step_name == "calculate_proration":

    # TODO 1: Extract plan details and start_date from step_input

    # TODO 2: If start_date is mid-cycle, calculate prorated charge using
proration_calculator

    # TODO 3: Calculate billing_cycle_anchor based on customer preference or plan
default

    # TODO 4: Determine first_billing_date considering trial periods

    # TODO 5: Return dict with prorated_amount, billing_anchor, first_billing_date

pass


elif step_name == "generate_invoice":

    # TODO 1: Create invoice line items for prorated charges, setup fees, trial
periods

    # TODO 2: Apply customer credit balance if available using
credit_balance_manager

    # TODO 3: Calculate taxes if applicable based on customer location

    # TODO 4: Generate invoice using invoice_generator.create_invoice()

    # TODO 5: Return dict with invoice_id, total_amount_cents, and
line_item_details

pass


elif step_name == "process_payment":

    # TODO 1: Extract invoice total and payment_method_id from step_input

    # TODO 2: For trial subscriptions, authorize payment method without charging

    # TODO 3: For paid subscriptions, charge payment method for invoice total

    # TODO 4: Handle payment failures with appropriate error codes

    # TODO 5: Return dict with payment_result, transaction_id, and payment_status
```

```
pass

elif step_name == "create_subscription":

    # TODO 1: Extract all subscription details from previous step outputs

    # TODO 2: Determine initial subscription status (trialing, active, incomplete)

    # TODO 3: Create subscription record using
subscription_engine.create_subscription()

    # TODO 4: Set next_billing_date based on trial period or billing cycle

    # TODO 5: Return dict with subscription_id, status, and next_billing_date

pass


elif step_name == "initialize_usage_tracking":

    # TODO 1: Extract subscription_id and plan details from step_input

    # TODO 2: Initialize usage allowances based on plan.usage_allowances

    # TODO 3: Set up usage quotas and rate limiting if configured

    # TODO 4: Create initial usage aggregation records for current billing period

    # TODO 5: Return dict with usage_tracking_initialized flag and quota_details

pass


else:

    raise ValueError(f"Unknown workflow step: {step_name}")


async def compensate_step(self, step_name: str, step_output: Dict[str, Any], execution: WorkflowExecution) -> None:

    """Rollback completed steps during failure recovery."""

    if step_name == "create_subscription":
```

```
# TODO: Cancel created subscription and mark as incomplete

subscription_id = step_output.get('subscription_id')

if subscription_id:

    await self.subscription_engine.cancel_subscription(
        subscription_id, immediate=True, reason="workflow_rollback"
    )

elif step_name == "process_payment":

    # TODO: Refund any charges that were processed

    payment_result = step_output.get('payment_result')

    if payment_result and payment_result.status == PaymentStatus.SUCCEEDED:

        await self.payment_processor.refund_payment(
            payment_result.payment_id, payment_result.amount,
            reason="subscription_creation_failed"
        )

elif step_name == "generate_invoice":

    # TODO: Mark invoice as void to prevent collection attempts

    invoice_id = step_output.get('invoice_id')

    if invoice_id:

        await self.invoice_generator.void_invoice(invoice_id,
reason="workflow_failed")
```

Billing Cycle Workflow Skeleton

```
# internal/workflows/billing_cycle.py                                PYTHON

class BillingCycleWorkflow(WorkflowExecutor):

    """Orchestrates monthly billing cycle processing."""

    def get_workflow_steps(self) -> List[str]:
        return [
            "identify_due_subscriptions",
            "aggregate_usage_data",
            "calculate_charges",
            "generate_invoices",
            "process_payments",
            "update_subscriptions",
            "reset_usage_tracking"
        ]

    @async_task
    async def execute_step(self, step_name: str, step_input: Dict[str, Any],
                          execution: WorkflowExecution) -> Dict[str, Any]:
        if step_name == "identify_due_subscriptions":
            # TODO 1: Query subscriptions where next_billing_date = current_date
            # TODO 2: Filter by status in [active, trialing] and exclude paused/cancelled
            # TODO 3: Group by customer timezone for accurate billing timing
            # TODO 4: Check for trial subscriptions ending today requiring activation
            # TODO 5: Return dict with subscription_ids list and timezone_groups
            pass
```

```

    elif step_name == "aggregate_usage_data":

        # TODO 1: For each subscription, get last_billing_date and current billing date

        # TODO 2: Call usage_tracker.aggregate_billing_period() for date range

        # TODO 3: Calculate overage amounts for usage exceeding plan allowances

        # TODO 4: Handle timezone-specific aggregation windows correctly

        # TODO 5: Return dict with usage_aggregations keyed by subscription_id

    pass

# Additional steps follow same pattern...

```

Milestone Checkpoints

Subscription Creation Flow Checkpoint: After implementing the subscription creation workflow, verify functionality with these tests:

1. **Successful Creation Test:** Create subscription with valid plan and payment method

- Expected: Subscription in `active` status, invoice generated and paid, usage tracking initialized
- Command: `python -m pytest tests/integration/test_subscription_creation_flow.py::test_successful_creation`

2. **Trial Subscription Test:** Create subscription with trial period

- Expected: Subscription in `trialing` status, payment method authorized but not charged
- Command: `python -m pytest tests/integration/test_subscription_creation_flow.py::test_trial_creation`

3. **Payment Failure Recovery:** Test subscription creation with failed payment

- Expected: Subscription in `incomplete` status, no access granted, retry scheduled
- Command: `python -m pytest tests/integration/test_subscription_creation_flow.py::test_payment_failure`

Billing Cycle Flow Checkpoint: Verify billing cycle processing with these scenarios:

1. **Standard Renewal:** Process renewal for active subscription with usage

- Expected: New invoice generated, payment processed, next billing date updated
- Command: `python scripts/run_billing_cycle.py --subscription-id <test_id>`

2. **Usage Overage Billing:** Process subscription with usage exceeding allowances

- Expected: Base charge plus overage charges, detailed line items in invoice
- Command: `python -m pytest tests/integration/test_billing_cycle_flow.py::test_usage_overage`

Plan Change Flow Checkpoint: Test plan change scenarios thoroughly:

1. **Immediate Upgrade:** Change to higher-tier plan with immediate effect

- Expected: Prorated charge applied, features activated immediately, next billing unchanged
- Command: `python -m pytest tests/integration/test_plan_change_flow.py::test_immediate_upgrade`

2. **Downgrade with Credit:** Change to lower-tier plan with credit calculation

- Expected: Credit applied to customer balance, features adjusted, future invoices reduced
- Command: `python -m pytest tests/integration/test_plan_change_flow.py::test_downgrade_with_credit`

Common Integration Issues

Symptom	Likely Cause	Diagnosis	Fix
Subscription creation hangs	Payment gateway timeout	Check payment processor logs for network timeouts	Add timeout configuration and retry logic
Billing cycle processes same subscription twice	Missing idempotency key or duplicate billing dates	Check for null idempotency_key in billing_executions table	Implement unique constraints on billing execution
Plan changes lose feature access	Race condition between plan update and feature check	Check timestamp ordering of plan_change and access_check events	Add version numbers to subscription feature cache
Usage aggregation returns zero	Timezone mismatch in billing period boundaries	Compare usage_event timestamps with billing_period_start/end	Normalize all timestamps to UTC in database
Proration calculations drift over time	Rounding errors accumulating across multiple changes	Track total_adjustments_cents field and check for drift	Use banker's rounding and periodic reconciliation

Error Handling and Edge Cases

Milestone(s): All milestones - provides robust failure recovery, consistency guarantees, and edge case handling essential for production billing operations across plan management (Milestone 1), subscription lifecycle (Milestone 2), proration (Milestone 3), and usage-based billing (Milestone 4)

Mental Model: Financial Fault Tolerance

Think of error handling in a subscription billing system like the safety systems in a nuclear power plant. In a power plant, every critical operation has multiple layers of protection: automatic shutdown systems, backup power supplies, containment structures, and emergency response procedures. Similarly, billing systems need multiple layers of protection because financial errors can be catastrophic - a single bug could overcharge thousands of customers or lose millions in revenue.

Just as a nuclear engineer designs for "defense in depth" with multiple independent safety systems, we must design billing systems with layered error handling: input validation prevents bad data from entering, transaction boundaries ensure atomicity, idempotency keys prevent duplicate operations, retry mechanisms handle transient failures, and compensation workflows can roll back complex multi-step operations. The goal is that no single point of failure can compromise the financial integrity of the system.

Like emergency response teams that regularly drill disaster scenarios, billing systems must be tested against failure modes that seem unlikely but would be devastating if they occurred. This means thinking through edge cases like leap years affecting billing dates, timezone changes during daylight saving transitions, and what happens when external payment systems become unavailable during peak billing periods.

Payment Failure Recovery

Payment failures are inevitable in subscription billing - credit cards expire, bank accounts run out of funds, and payment processors experience outages. The key to graceful payment failure recovery is implementing a systematic **dunning management** process that maximizes revenue recovery while maintaining positive customer relationships.

The foundation of payment failure recovery is the **grace period** concept. When a payment fails, the subscription doesn't immediately terminate. Instead, it transitions to a `past_due` status where the customer retains access to the service while the system attempts recovery. This approach recognizes that many payment failures are temporary - a declined card might work again in a few hours after the customer resolves a temporary hold.

Decision: Dunning Management Strategy

- **Context:** Payment failures require systematic retry logic with escalating actions to maximize recovery while avoiding customer frustration
- **Options Considered:** Immediate cancellation, fixed retry schedule, exponential backoff with customer communication
- **Decision:** Multi-stage dunning with smart retry timing and proactive customer communication
- **Rationale:** Balances revenue recovery (fixed schedules recover more revenue) with customer experience (exponential backoff reduces payment spam)
- **Consequences:** Requires complex state management but significantly improves payment recovery rates and customer satisfaction

The dunning process follows a structured escalation pattern designed around customer psychology and payment system behaviors:

Dunning Stage	Timing	Action	Purpose	Success Rate
Immediate Retry	1 hour after failure	Retry payment with same method	Catch temporary declines	15-20%
Early Recovery	Day 1, 3, 5	Retry + email notification	Allow customer self-service	25-30%
Active Recovery	Day 7, 10	Retry + access restriction warnings	Create urgency	20-25%
Final Recovery	Day 14	Final retry + suspension notice	Last chance recovery	10-15%
Suspension	Day 17	Suspend service, retain data	Minimize churn	5-10% monthly recovery

Smart retry timing adapts to failure types because different decline reasons have different optimal retry patterns. A "insufficient funds" decline might succeed after payday, while "card expired" requires customer action. The system tracks decline codes from the payment gateway and adjusts retry schedules accordingly:

Decline Code Analysis:

- Insufficient Funds → Retry after 3-5 days (payday cycle)
- Expired Card → Email immediately, retry weekly
- Fraud Prevention → Retry hourly for first day, then daily
- Processing Error → Exponential backoff starting at 1 hour

Dunning Action execution must be idempotent because network failures can cause duplicate webhook deliveries. Each dunning attempt receives a unique idempotency key combining the subscription ID, invoice ID, and attempt number. The system tracks all dunning attempts in an audit trail to prevent duplicate charges and ensure compliance with payment regulations.

Customer communication during dunning is critical for maintaining relationships. The system generates contextual messages based on the failure reason and customer history. High-value customers might receive phone call triggers, while newer customers get educational emails about updating payment methods. The key insight is that dunning is a customer success opportunity, not just a revenue recovery mechanism.

The critical principle in payment failure recovery is graduated response - start gently with technical retries, escalate to customer communication, and only restrict access as a last resort. Customers often don't realize their payment failed until they receive a notification.

Payment method fallback provides an additional recovery layer. When a customer has multiple payment methods on file, the system automatically attempts the secondary method after the primary fails. This requires careful sequencing to avoid double-charging if the primary payment succeeds on retry after the secondary was already charged.

Data Consistency and Concurrency

Subscription billing systems must maintain strict **financial precision** across distributed operations while handling concurrent access from multiple sources: customer self-service portals, admin interfaces, automated billing cycles, and webhook processing from payment gateways. The challenge is ensuring that financial state remains consistent even when multiple operations attempt to modify the same subscription simultaneously.

The foundation of consistency in billing systems is the **transactional outbox** pattern. Every financial operation - subscription creation, plan changes, usage recording, payment processing - executes within a single database transaction that both updates business entities and writes events to an outbox table. This ensures that the business state change and the corresponding event publication are atomic, preventing the common failure mode where a database update succeeds but the downstream notification fails.

Operation Type	Consistency Requirements	Isolation Level	Concurrent Conflicts
Subscription Creation	Customer must exist, plan must be valid	SERIALIZABLE	Duplicate customer subscriptions
Plan Changes	No overlapping changes in billing cycle	SERIALIZABLE	Mid-cycle plan change races
Usage Recording	Idempotent event ingestion	READ_COMMITTED	Duplicate event submission
Payment Processing	Invoice amount must match charges	SERIALIZABLE	Concurrent payment attempts
Credit Application	Balance must not go negative	SERIALIZABLE	Credit balance races

Optimistic concurrency control prevents lost updates when multiple processes modify subscription state simultaneously. Each financial entity includes a version number that increments with every update. Operations read the current version, perform their calculations, and attempt to update only if the version matches. If the version has changed, indicating another process modified the entity, the operation retries with fresh data.

The subscription state machine enforces valid transitions through database constraints and application logic. Concurrent state changes are serialized using database row locks on the subscription record. For example, if an automated renewal process attempts to transition a subscription from `active` to `past_due` due to payment failure, while simultaneously an admin user tries to cancel the subscription, the database ensures only one transition succeeds and the other receives a conflict error that triggers retry logic.

Concurrency Control Example:

1. Billing cycle starts renewal for subscription_123 (version 15)
2. Admin user starts cancellation for subscription_123 (version 15)
3. Billing update: `SET status='past_due', version=16 WHERE id='123' AND version=15`
4. Admin update: `SET status='cancelled', version=16 WHERE id='123' AND version=15`
5. First update succeeds, second fails (version already 16)
6. Admin operation retries with version 16, sees `past_due` status
7. Admin logic decides: cancel `past_due` subscription immediately

Credit balance management requires special attention to race conditions because credits can be applied and consumed concurrently. The system uses database-level atomic operations for credit adjustments, implemented through stored procedures or application-level locking. Credit consumption operations always check available balance within the same transaction that applies the credit to prevent negative balances.

Event ordering and idempotency prevent duplicate processing when webhook deliveries arrive out of order or multiple times. Each webhook event includes a gateway-provided event ID and timestamp. The system maintains a processed events table with unique constraints on the gateway event ID, ensuring duplicate webhook deliveries are silently ignored rather than causing double-processing.

Decision: Event Processing Architecture

- **Context:** Webhook events can arrive out of order, be delivered multiple times, or be lost entirely, requiring robust event processing
- **Options Considered:** Direct processing, event sourcing with replay, transactional outbox with deduplication
- **Decision:** Transactional outbox with idempotency keys and event ordering by timestamp
- **Rationale:** Provides exactly-once processing guarantees while maintaining audit trail and enabling replay for debugging
- **Consequences:** Additional storage overhead but eliminates duplicate payment processing and enables reliable event replay

Distributed transaction management across multiple services uses the **saga pattern** rather than two-phase commit. Complex operations like subscription creation span multiple components: customer validation, plan entitlement setup, payment method verification, and initial invoice generation. Each step in the saga can complete independently, with compensation actions defined to roll back completed steps if later steps fail.

The saga coordinator tracks workflow progress and handles partial failures by executing compensation steps in reverse order. For example, if subscription creation fails during payment method verification after successfully creating the subscription record and setting up entitlements, the compensation workflow removes the entitlements and marks the subscription as `incomplete` rather than leaving it in an inconsistent state.

Billing Edge Cases

Billing systems encounter numerous edge cases related to time, calendar arithmetic, and timezone handling that can cause subtle but expensive bugs. These edge cases often surface during specific calendar events like leap years, daylight saving transitions, and month-end billing cycles, making them difficult to catch in normal testing but critical to handle correctly in production.

Calendar arithmetic presents the most common source of billing edge cases. The **billing anchor** concept helps maintain consistent billing dates, but month overflow situations require careful handling. When a subscription starts on January 31st with monthly billing, the naive approach of adding one month results in February 31st, which doesn't exist. The system must define consistent rules for handling these overflows:

Billing Anchor	Next Billing Date	Rule Applied	Alternative Approaches
January 31	February 28 (non-leap), February 29 (leap)	End-of-month clamping	Use February 28 always, Use March 3
March 31	April 30	Month-end clamping	Use April 31 → May 1
February 29 (leap year)	March 29 (following year)	Day preservation when possible	Always use March 28, Use March 1

The chosen approach is **anchor day preservation** with end-of-month clamping. If the anchor day exists in the target month, use it. If not, use the last day of the target month. This provides predictable billing dates that customers can understand while avoiding the complexity of variable-length billing periods.

Timezone handling becomes complex when customers can change their billing timezone or when daylight saving transitions affect billing schedules. The system stores all timestamps in UTC and converts to customer timezone only for display purposes. Billing calculations always use UTC to ensure consistency, but customer communication uses local time to avoid confusion.

Daylight saving transitions can cause apparent billing time shifts. A customer in New York with 9 AM billing might see their next bill generated at 8 AM local time after the spring forward transition. To avoid customer confusion, the system communicates billing times in terms of "billing date" rather than specific times, and processes all billing for a given date in a single batch operation.

The cardinal rule of billing timezone handling is: store in UTC, calculate in UTC, display in customer timezone. Never perform financial calculations using local time - daylight saving transitions can cause billing to run twice or be skipped entirely.

Leap year handling affects annual billing cycles and usage aggregation periods. February 29th poses special challenges for anniversary-based billing. A customer who subscribes on February 29th during a leap year needs a consistent anniversary date in non-leap years. The system uses February 28th as the anniversary in non-leap years, with clear customer communication about this adjustment.

Usage aggregation during leap years must account for the extra day in February when calculating monthly averages or comparing year-over-year usage patterns. The system normalizes usage metrics by billing period length to ensure fair comparisons across different month lengths.

Currency precision and rounding requires careful attention to avoid accumulating errors across many customers. The system uses integer arithmetic with the smallest currency unit (cents for USD, pence for GBP) to avoid floating-point precision issues. All monetary calculations use the `Money` type with explicit rounding rules applied at specific points in the calculation pipeline.

Proration calculations are particularly susceptible to rounding errors because they involve division operations. The system applies rounding only at the final step of each calculation, maintaining full precision through intermediate steps. When proration results in fractional cents, the system uses "round half to even" (banker's rounding) to prevent systematic bias in either customer or company favor.

Rounding Scenario	Amount Before Rounding	Rounded Amount	Rule Applied
Monthly proration	\$29.996666...	\$30.00	Round up at 0.5 cents
Daily proration	\$1.235	\$1.24	Round half to even (odd result)
Quantity-based proration	\$15.245	\$15.24	Round half to even (even result)
Usage overage	\$0.001234	\$0.00	Minimum charge threshold

Cross-month billing cycles create edge cases when subscription events occur near month boundaries. A subscription cancelled on the last day of the month might have its final invoice generated in the following month, affecting revenue reporting and customer communication timing. The system maintains strict separation between billing period boundaries and invoice generation dates to ensure accurate financial reporting.

Retroactive adjustments handle situations where billing corrections must be applied after invoices have been generated. This occurs when usage data arrives late, pricing errors are discovered, or customer service approves billing adjustments. The system creates adjustment line items on future invoices rather than modifying historical invoices, maintaining audit trail integrity while ensuring customers see the corrections.

Usage event late arrival requires special handling because aggregation windows must close to generate invoices, but some events may arrive after the window closes. The system defines a "late arrival window" (typically 7 days) during which late events are accepted and applied as adjustments to the following invoice. Events arriving after this window are logged but not applied to avoid indefinite billing uncertainty.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
Error Tracking	Python logging + file rotation	Sentry or Datadog APM with structured logging
Circuit Breaker	Custom retry with exponential backoff	Tenacity library with circuit breaker patterns
Database Transactions	PostgreSQL with explicit transaction blocks	SQLAlchemy with session management and savepoints
Webhook Processing	Flask/FastAPI with request validation	Celery task queue with Redis for webhook processing
Event Outbox	Database table with background polling	Debezium CDC with Kafka for reliable event streaming
Concurrency Control	Database row locks with version columns	Distributed locks with Redis or ZooKeeper

Recommended File Structure

```
billing_system/
  src/
    error_handling/
      __init__.py
      exceptions.py          ← Custom exception hierarchy
      retry_policies.py     ← Retry strategies and circuit breakers
      transaction_manager.py ← Database transaction utilities
    dunning/
      __init__.py
      dunning_engine.py     ← Payment failure recovery logic
      dunning_actions.py    ← Email, restriction, retry actions
      dunning_scheduler.py   ← Retry timing and escalation
    concurrency/
      __init__.py
      locks.py              ← Optimistic concurrency control
      outbox.py              ← Transactional outbox pattern
      saga_coordinator.py   ← Distributed transaction management
    edge_cases/
      __init__.py
      calendar_utils.py     ← Timezone and date arithmetic
      currency_precision.py ← Money handling and rounding
      billing_calendar.py   ← Billing anchor and cycle calculation
  tests/
    test_error_scenarios/
    test_concurrency/
    test_edge_cases/       ← Payment failure simulation tests
                           ← Race condition and deadlock tests
                           ← Calendar edge case test suite
```

Infrastructure Starter Code

Custom Exception Hierarchy

```
# src/error_handling/exceptions.py

from typing import Optional, Dict, Any

from enum import Enum

import logging

class BillingErrorType(Enum):

    PAYMENT_FAILED = "payment_failed"

    INSUFFICIENT_CREDIT = "insufficient_credit"

    INVALID_SUBSCRIPTION_STATE = "invalid_subscription_state"

    PRORATION_ERROR = "proration_error"

    USAGE_VALIDATION_ERROR = "usage_validation_error"

    CONCURRENCY_CONFLICT = "concurrency_conflict"

    EXTERNAL_SERVICE_ERROR = "external_service_error"

class BillingException(Exception):

    """Base exception for all billing system errors with structured error information."""

    def __init__(

        self,
        message: str,
        error_type: BillingErrorType,
        entity_id: Optional[str] = None,
        metadata: Optional[Dict[str, Any]] = None,
        cause: Optional[Exception] = None
    ):

        super().__init__(message)

        self.error_type = error_type

        self.entity_id = entity_id
```

```
        self.metadata = metadata or {}

        self.cause = cause


def to_dict(self) -> Dict[str, Any]:
    return {

        "error_type": self.error_type.value,
        "message": str(self),
        "entity_id": self.entity_id,
        "metadata": self.metadata
    }

class PaymentFailedException(BillingException):

    def __init__(self, message: str, payment_id: str, decline_code: Optional[str] = None):
        metadata = {"payment_id": payment_id}

        if decline_code:
            metadata["decline_code"] = decline_code

        super().__init__(message, BillingErrorType.PAYMENT_FAILED, payment_id, metadata)

class ConcurrencyConflictException(BillingException):

    def __init__(self, message: str, entity_type: str, entity_id: str, expected_version: int, actual_version: int):
        metadata = {

            "entity_type": entity_type,
            "expected_version": expected_version,
            "actual_version": actual_version
        }

        super().__init__(message, BillingErrorType.CONCURRENCY_CONFLICT, entity_id, metadata)
```

```
class SubscriptionStateException(BillingException):

    def __init__(self, message: str, subscription_id: str, current_state: str,
attempted_transition: str):

        metadata = {

            "current_state": current_state,

            "attempted_transition": attempted_transition

        }

        super().__init__(message, BillingErrorType.INVALID_SUBSCRIPTION_STATE,
subscription_id, metadata)
```

Retry Policy Implementation

```
# src/error_handling/retry_policies.py

import time

import random

import logging

from typing import Optional, Callable, Type, Tuple

from functools import wraps

from dataclasses import dataclass

from .exceptions import BillingException, BillingErrorType


@dataclass

class RetryConfig:

    max_attempts: int = 3

    base_delay: float = 1.0

    max_delay: float = 60.0

    exponential_base: float = 2.0

    jitter: bool = True

    retryable_exceptions: Tuple[Type[Exception], ...] = (Exception,)

    non_retryable_error_types: Tuple[BillingErrorType, ...] = (

        BillingErrorType.INVALID_SUBSCRIPTION_STATE,
        BillingErrorType.INSUFFICIENT_CREDIT
    )



class CircuitBreaker:

    """Circuit breaker implementation to prevent cascading failures."""

    def __init__(self, failure_threshold: int = 5, timeout: float = 60.0):

        self.failure_threshold = failure_threshold

        self.timeout = timeout
```

```
    self.failure_count = 0

    self.last_failure_time = None

    self.state = "closed" # closed, open, half_open


def is_available(self) -> bool:
    if self.state == "closed":
        return True
    elif self.state == "open":
        if time.time() - self.last_failure_time >= self.timeout:
            self.state = "half_open"
        return True
    return False
else: # half_open
    return True


def record_success(self):
    self.failure_count = 0
    self.state = "closed"


def record_failure(self):
    self.failure_count += 1
    self.last_failure_time = time.time()
    if self.failure_count >= self.failure_threshold:
        self.state = "open"


def retry_with_backoff(config: RetryConfig = None):
    """Decorator for automatic retry with exponential backoff."""

```

```
if config is None:

    config = RetryConfig()


def decorator(func):

    @wraps(func)

    def wrapper(*args, **kwargs):

        last_exception = None

        for attempt in range(config.max_attempts):

            try:

                return func(*args, **kwargs)

            except Exception as e:

                last_exception = e

                # Check if exception is retryable

                if not isinstance(e, config.retryable_exceptions):

                    raise

                # Check if billing error type is non-retryable

                if isinstance(e, BillingException) and e.error_type in config.non_retryable_error_types:

                    raise

            # Don't sleep on last attempt

            if attempt == config.max_attempts - 1:

                break
```

```
# Calculate delay with exponential backoff and jitter

delay = min(
    config.base_delay * (config.exponential_base ** attempt),
    config.max_delay
)

if config.jitter:
    delay = delay * (0.5 + random.random() * 0.5)

logging.warning(f"Attempt {attempt + 1} failed, retrying in
{delay:.2f}s: {e}")

time.sleep(delay)

raise last_exception

return wrapper

return decorator
```

Transaction Manager

```
# src/error_handling/transaction_manager.py
```

PYTHON

```
import logging

from contextlib import contextmanager

from typing import Optional, Any, Dict

from sqlalchemy import create_engine

from sqlalchemy.orm import sessionmaker, Session

from sqlalchemy.exc import SQLAlchemyError

from .exceptions import BillingException, BillingErrorType


class TransactionManager:

    """Manages database transactions with proper error handling and rollback."""

    def __init__(self, database_url: str):

        self.engine = create_engine(database_url)

        self.SessionLocal = sessionmaker(bind=self.engine)

    @contextmanager

    def transaction(self, isolation_level: Optional[str] = None):

        """Database transaction context manager with automatic rollback on errors."""

        session = self.SessionLocal()

        if isolation_level:

            session.connection(execution_options={"isolation_level": isolation_level})

        try:

            yield session

            session.commit()

        except Exception as e:
```

```
        logging.info("Transaction committed successfully")

    except SQLAlchemyError as e:

        session.rollback()

        logging.error(f"Database error, transaction rolled back: {e}")

        raise BillingException(
            f"Database transaction failed: {str(e)}",
            BillingErrorType.EXTERNAL_SERVICE_ERROR,
            cause=e
        )

    except Exception as e:

        session.rollback()

        logging.error(f"Unexpected error, transaction rolled back: {e}")

        raise

    finally:

        session.close()

def create_savepoint(self, session: Session, savepoint_name: str):

    """Create named savepoint for partial rollback within transaction."""

    try:

        session.begin_nested()

        return savepoint_name

    except SQLAlchemyError as e:

        raise BillingException(
            f"Failed to create savepoint {savepoint_name}: {str(e)}",
            BillingErrorType.EXTERNAL_SERVICE_ERROR,
            cause=e
        )
```

)

Core Logic Skeleton Code

Dunning Engine Implementation

```
# src/dunning/dunning_engine.py

from datetime import datetime, timedelta

from typing import List, Optional, Dict, Any

from dataclasses import dataclass

from enum import Enum

from ..models import Subscription, Invoice, Payment, PaymentStatus

from ..error_handling.exceptions import BillingException, BillingErrorType

from ..error_handling.retry_policies import retry_with_backoff, RetryConfig


class DunningStage(Enum):

    IMMEDIATE_RETRY = "immediate_retry"

    EARLY_RECOVERY = "early_recovery"

    ACTIVE_RECOVERY = "active_recovery"

    FINAL_RECOVERY = "final_recovery"

    SUSPENDED = "suspended"


@dataclass

class DunningAttempt:

    attempt_id: str

    subscription_id: str

    invoice_id: str

    stage: DunningStage

    scheduled_at: datetime

    completed_at: Optional[datetime]

    success: bool

    decline_code: Optional[str]

    next_attempt_at: Optional[datetime]
```

```
class DunningEngine:

    """Manages payment failure recovery with graduated escalation."""

    def __init__(self, payment_processor, notification_service, transaction_manager):

        self.payment_processor = payment_processor

        self.notification_service = notification_service

        self.transaction_manager = transaction_manager

    def handle_payment_failure(
            self,
            subscription_id: str,
            invoice_id: str,
            failure_reason: str,
            idempotency_key: str
        ) -> None:

        """Initiate dunning process for failed payment."""

        # TODO 1: Check if dunning already started for this invoice (idempotency)

        # TODO 2: Analyze failure reason to determine appropriate retry strategy

        # TODO 3: Transition subscription to past_due status with grace period

        # TODO 4: Schedule immediate retry attempt based on decline code

        # TODO 5: Send initial customer notification about payment failure

        # TODO 6: Create dunning attempt record with tracking information

        # Hint: Different decline codes need different retry timing

        pass

    def process_dunning_retry(self, subscription_id: str) -> bool:

        """Execute next dunning attempt for subscription."""
```

```
# TODO 1: Load current dunning state and determine next stage

# TODO 2: Check if subscription is still eligible for dunning

# TODO 3: Attempt payment retry using stored payment method

# TODO 4: Process retry result and update dunning state

# TODO 5: Schedule next attempt or escalate based on result

# TODO 6: Send appropriate customer communication for stage

# TODO 7: Update subscription status if recovery successful

# Hint: Use exponential backoff for technical failures, fixed schedule for customer
issues

pass


def escalate_dunning_stage(self, subscription_id: str, current_stage: DunningStage) ->
DunningStage:

    """Determine next dunning stage based on current stage and elapsed time."""

    # TODO 1: Calculate days since initial payment failure

    # TODO 2: Apply dunning policy rules to determine next stage

    # TODO 3: Check for any customer-specific dunning overrides

    # TODO 4: Return appropriate next stage or suspension

    # Hint: Different customer tiers may have different dunning schedules

    pass


@retry_with_backoff(RetryConfig(max_attempts=3, base_delay=2.0))

def attempt_payment_recovery(self, subscription_id: str, payment_method_id: str) ->
bool:

    """Attempt payment retry with the customer's payment method."""

    # TODO 1: Validate subscription is in past_due status

    # TODO 2: Calculate amount due including any late fees

    # TODO 3: Call payment processor with idempotency protection
```

```
# TODO 4: Handle payment success/failure and update records

# TODO 5: Apply any credits or adjustments to the payment

# Hint: Always use unique idempotency keys for each retry attempt

pass
```

Concurrency Control Implementation

```
# src/concurrency/locks.py
```

PYTHON

```
from typing import Optional, Any, Dict, Type, TypeVar

from sqlalchemy.orm import Session

from sqlalchemy.exc import IntegrityError

from ..models import BaseEntity

from ..error_handling.exceptions import ConcurrencyConflictException

T = TypeVar('T', bound=BaseEntity)

class OptimisticLockManager:

    """Manages optimistic concurrency control using entity version numbers."""

    def __init__(self, session: Session):

        self.session = session


    def load_with_lock(self, entity_class: Type[T], entity_id: str) -> Optional[T]:

        """Load entity with current version for optimistic locking."""

        # TODO 1: Query entity by ID and return with current version

        # TODO 2: Return None if entity doesn't exist

        # TODO 3: Log entity load for debugging concurrency issues

        # Hint: Always load the version column for comparison

        pass


    def save_with_version_check(self, entity: T) -> T:

        """Save entity with optimistic lock version checking."""

        # TODO 1: Increment entity version number before save

        # TODO 2: Attempt database update with WHERE version = old_version

        # TODO 3: Check if update affected any rows (version conflict)
```

```
# TODO 4: Raise ConcurrencyConflictException if version mismatch

# TODO 5: Return updated entity with new version

# TODO 6: Handle database constraint violations gracefully

# Hint: Use UPDATE ... WHERE id = ? AND version = ? to detect conflicts

pass

def retry_on_conflict(self, operation_func, max_retries: int = 3):

    """Retry operation on optimistic lock conflicts."""

    # TODO 1: Execute operation function with fresh entity load

    # TODO 2: Catch ConcurrencyConflictException and retry

    # TODO 3: Implement exponential backoff between retries

    # TODO 4: Re-raise exception after max retries exceeded

    # TODO 5: Log retry attempts for monitoring

    # Hint: Reload entity data before each retry attempt

    pass
```

Billing Edge Case Utilities

```
# src/edge_cases/billing_calendar.py
```

PYTHON

```
from datetime import datetime, date, timedelta
from typing import Optional
from dateutil.relativedelta import relativedelta
import pytz
from ..models import BillingInterval

class BillingCalendar:

    """Handles billing date calculations with proper edge case handling."""

    def calculate_next_billing_date(
        self,
        current_date: date,
        billing_interval: BillingInterval,
        billing_anchor: int,
        timezone: Optional[str] = None
    ) -> date:
        """Calculate next billing date handling month overflow and leap years."""

        # TODO 1: Handle timezone conversion from UTC to customer timezone
        # TODO 2: Apply billing interval (monthly, yearly) using relativedelta
        # TODO 3: Handle month overflow (Jan 31 + 1 month = Feb 28/29)
        # TODO 4: Handle leap year edge cases for February 29 anchors
        # TODO 5: Ensure billing date doesn't go backwards due to DST
        # TODO 6: Convert result back to UTC for storage
        # Hint: Use end-of-month clamping for month overflow situations
        pass
```

```
def calculate_proration_factor(  
    self,  
    change_date: datetime,  
    period_start: datetime,  
    period_end: datetime  
) -> float:  
  
    """Calculate time-based proration factor for partial billing periods."""  
  
    # TODO 1: Validate that change_date falls within billing period  
  
    # TODO 2: Calculate total period length in seconds for precision  
  
    # TODO 3: Calculate remaining period length from change date  
  
    # TODO 4: Return factor as remaining_time / total_time  
  
    # TODO 5: Handle edge case where change_date equals period boundaries  
  
    # TODO 6: Ensure factor is between 0.0 and 1.0  
  
    # Hint: Use seconds for calculation to avoid leap second issues  
  
    pass  
  
  
def handle_leap_year_anniversary(self, original_date: date, target_year: int) -> date:  
  
    """Handle leap year edge cases for annual billing anniversaries."""  
  
    # TODO 1: Check if original date is February 29  
  
    # TODO 2: Check if target year is a leap year  
  
    # TODO 3: If Feb 29 -> non-leap year, use Feb 28  
  
    # TODO 4: If Feb 29 -> leap year, preserve Feb 29  
  
    # TODO 5: Handle other leap year boundary conditions  
  
    # TODO 6: Return adjusted anniversary date  
  
    # Hint: Use calendar.isleap() to check leap year status  
  
    pass
```

Milestone Checkpoints

Payment Failure Recovery Validation:

- Start subscription with valid payment method
- Simulate payment failure by marking payment method as declined
- Verify subscription transitions to `past_due` status within grace period
- Check that retry attempts are scheduled according to dunning policy
- Confirm customer notifications are sent at appropriate intervals
- Validate that successful retry restores subscription to `active` status

Concurrency Control Testing:

- Start two concurrent plan change operations on same subscription
- Verify that only one operation succeeds and other receives conflict error
- Check that failed operation retries with fresh entity version
- Confirm that final state is consistent with last successful operation
- Test credit balance race conditions with concurrent consumption

Edge Case Handling Verification:

- Create subscription with January 31 billing anchor
- Verify February billing date is February 28 (or 29 in leap year)
- Test timezone changes during daylight saving transitions
- Confirm proration calculations handle fractional cents correctly
- Validate leap year anniversary handling for February 29 subscriptions

Common Pitfalls and Debugging

Symptom	Likely Cause	How to Diagnose	Fix
Duplicate payment charges	Webhook processing without idempotency	Check payment_events table for duplicate gateway_event_id	Add unique constraint on gateway_event_id
Subscription stuck in past_due	Dunning retry logic not scheduling	Check dunning_attempts table for next_attempt_at values	Fix dunning scheduler cron job
Proration amounts off by cents	Floating point precision errors	Log intermediate calculation values	Use integer cent arithmetic throughout
Billing dates drift over time	Month overflow not handled consistently	Compare expected vs actual billing dates	Implement end-of-month clamping rule
Credit balance goes negative	Race condition in credit consumption	Check for concurrent credit applications	Add database constraint on credit balance ≥ 0
Payment retries spam customer	Circuit breaker not working	Monitor payment attempt frequency	Implement exponential backoff with max delay

⚠ Pitfall: Payment Webhook Replay Processing the same webhook event multiple times can cause duplicate charges or state transitions. Always check for existing processing records using the gateway event ID before processing webhook data.

⚠ Pitfall: Timezone Arithmetic in Billing Performing billing date calculations in customer timezone can cause billing to run twice or be skipped during daylight saving transitions. Always calculate in UTC and convert only for display.

⚠ Pitfall: Optimistic Lock Retry Storms Under high concurrency, optimistic lock conflicts can cause retry storms. Implement exponential backoff and maximum retry limits to prevent system overload.

Testing Strategy and Validation

Milestone(s): All milestones - defines comprehensive testing approaches to validate billing logic correctness, integration reliability, and milestone completion across plan management (Milestone 1), subscription lifecycle (Milestone 2), proration calculations (Milestone 3), and usage-based billing (Milestone 4)

Mental Model: The Financial Audit Trail

Think of testing a subscription billing system like conducting a financial audit of a complex business. Just as an auditor must verify every transaction, reconcile every account, and trace money flows from source to destination, billing system testing requires meticulous validation at multiple levels. Unit tests are like checking

individual ledger entries for mathematical accuracy - ensuring each proration calculation, usage aggregation, and pricing computation produces the correct result down to the cent. Integration tests resemble departmental audits, verifying that when the billing department sends an invoice to accounts receivable, the payment flows correctly through the entire organization. End-to-end tests are like full company audits, following a customer's entire financial journey from subscription signup through monthly billing cycles to final cancellation.

The stakes are equally high in both scenarios. A single rounding error in proration calculations, multiplied across thousands of customers, can result in significant revenue loss or regulatory violations. A failed payment webhook that goes unnoticed can leave subscriptions in inconsistent states, causing customer service nightmares. Just as auditors use sampling techniques to validate large datasets, our testing strategy employs property-based testing to verify billing logic across thousands of generated scenarios, catching edge cases that manual test writing might miss.

Billing Logic Unit Tests

Unit testing billing logic requires mathematical precision and comprehensive edge case coverage. Unlike typical application logic that might tolerate minor inconsistencies, billing calculations must be absolutely correct - every penny must be accounted for, and rounding must be deterministic and consistent across all operations.

Proration Calculation Testing

Proration testing focuses on the mathematical accuracy of partial billing calculations. These tests must verify that time-based proration produces correct results across various billing intervals, calendar edge cases, and currency precision requirements.

The core proration test suite validates the fundamental mathematical properties that all proration calculations must satisfy:

Test Category	Test Cases	Validation Criteria
Basic Time Proration	Full period, half period, single day	Proration factor calculation accuracy
Currency Precision	Multiple currencies, rounding edge cases	No precision loss, consistent rounding
Calendar Edge Cases	Month overflow, leap years, timezone boundaries	Correct day counting, timezone handling
Plan Change Scenarios	Upgrade, downgrade, same-tier changes	Credit/charge amounts match expected
Quantity-Based Changes	Seat increases, usage tier changes	Per-unit proration accuracy

Property-based testing proves invaluable for proration validation. Rather than manually crafting test cases, property-based tests generate thousands of random scenarios and verify that mathematical invariants hold:

- **Symmetry Property:** Upgrading then immediately downgrading should result in zero net charge (minus any rounding)
- **Additivity Property:** Proration for multiple changes within a period should equal the sum of individual prorations
- **Monotonicity Property:** Larger plan changes should never result in smaller absolute proration amounts
- **Boundary Property:** Proration at period start should equal full charge, proration at period end should equal zero charge

The test data generation must cover realistic business scenarios while exploring mathematical edge cases. Time periods should span various billing intervals (monthly, quarterly, annual), change dates should include first/last days of billing cycles, and currency amounts should test precision boundaries.

Usage Aggregation Testing

Usage aggregation testing validates the mathematical accuracy of usage event processing and billing calculations. These tests must ensure that usage events aggregate correctly across time boundaries, duplicate events are properly deduplicated, and overage charges calculate accurately.

Aggregation Scenario	Test Coverage	Expected Behavior
Event Deduplication	Duplicate events with same idempotency key	Only single event counted
Time Boundary Handling	Events at billing period edges	Correct period assignment
Usage Type Separation	Multiple metric types per customer	Isolated aggregation per metric
Quantity Precision	Fractional usage quantities	Accurate decimal handling
Overage Calculations	Usage beyond plan allowances	Correct tiered rate application

Time series testing validates usage aggregation across billing period boundaries. These tests generate usage events with carefully controlled timestamps to verify that aggregation windows capture the correct events. Edge cases include events submitted exactly at billing period boundaries, events with future timestamps (clock skew), and events submitted after billing period close.

Idempotency verification ensures that reprocessing usage events produces identical aggregation results. Tests submit the same usage batch multiple times with identical idempotency keys and verify that total usage quantities remain unchanged. This validation protects against double-billing scenarios in production.

Pricing Model Testing

Pricing model testing validates the mathematical accuracy of charge calculations across different pricing structures. Each pricing model (flat-rate, tiered, volume-based) has distinct calculation logic that requires specialized test coverage.

Pricing Model	Test Scenarios	Mathematical Validation
Flat-Rate	Base charges, currency conversion	Fixed amount accuracy
Tiered Pricing	Single tier, multi-tier, boundary cases	Per-tier calculation accuracy
Volume Pricing	Usage below/above thresholds	Rate change threshold accuracy
Per-Seat Pricing	Seat changes, proration	Quantity-based calculation
Usage-Based	Allowances, overages, tier boundaries	Complex usage charge accuracy

Tiered pricing edge cases require special attention because they involve complex mathematical logic. Tests must verify calculations when usage spans multiple pricing tiers, ensuring that each tier applies its rate only to the usage quantity within that tier's boundaries. Boundary conditions (usage exactly at tier thresholds) must be tested exhaustively.

Feature Entitlement Testing

Feature entitlement testing validates the business logic that determines customer access to features based on their subscription plan. These tests ensure that feature flags, usage quotas, and access controls operate correctly across plan changes and subscription state transitions.

Entitlement Type	Test Coverage	Validation Points
Boolean Features	Feature enabled/disabled by plan	Correct true/false determination
Quota Limits	Usage tracking against plan limits	Accurate quota enforcement
Plan Hierarchies	Feature inheritance in plan families	Correct access level determination
Time-Based Access	Trial periods, grace periods	Time-sensitive feature access

Integration and End-to-End Tests

Integration testing validates the interaction between billing system components and external services. Unlike unit tests that isolate individual calculations, integration tests verify that data flows correctly between components and that system-wide business processes complete successfully.

Payment Gateway Integration Testing

Payment gateway integration testing validates the complete payment flow from charge creation through webhook processing. These tests must handle asynchronous payment processing, webhook delivery reliability, and failure scenario recovery.

The integration test environment requires a payment gateway test mode that simulates various payment scenarios without processing real money. Test cases must cover successful payments, failed payments with different decline reasons, and webhook delivery edge cases.

Integration Scenario	Test Setup	Success Criteria
Successful Payment Flow	Valid payment method, sufficient funds	Payment completed, subscription activated
Payment Failure Handling	Invalid payment method or declined card	Dunning process initiated, customer notified
Webhook Processing	Payment status change webhooks	Subscription state updated correctly
Idempotency Protection	Duplicate webhook delivery	No duplicate processing occurred
Timeout Scenarios	Delayed webhook delivery	Proper retry and recovery behavior

Webhook reliability testing focuses on the asynchronous nature of payment notifications. Tests must simulate webhook delivery failures, duplicate deliveries, and out-of-order delivery to ensure the system maintains consistency under adverse network conditions.

The test suite uses webhook simulation rather than depending on actual payment gateway webhook delivery. This approach provides deterministic test execution while covering the same code paths that process production webhooks.

Subscription Lifecycle Integration Testing

Subscription lifecycle integration testing validates end-to-end subscription operations that span multiple system components. These tests verify that subscription state changes trigger appropriate downstream actions and maintain data consistency across components.

Lifecycle Operation	Component Integration	Validation Points
Subscription Creation	Plan validation, payment setup, activation	All components reflect active subscription
Plan Changes	Proration calculation, billing adjustment	Correct charges applied to next invoice
Renewal Processing	Usage aggregation, invoice generation	Complete billing cycle execution
Cancellation	Service termination, final billing	Proper cleanup and final charges

Billing cycle integration testing validates the monthly billing process that aggregates usage, applies proration adjustments, generates invoices, and processes payments. This complex workflow requires coordination between multiple components and represents the most critical business process.

The test creates a controlled billing scenario with known usage events, plan changes, and payment methods. It then executes the complete billing cycle and validates that the generated invoice contains the correct line items with accurate amounts.

Database Transaction Testing

Database transaction testing validates the consistency guarantees provided by the system's transaction management. These tests verify that multi-step operations either complete entirely or roll back completely, preventing partial state corruption.

Transaction Scenario	Test Coverage	Consistency Validation
Plan Change Rollback	Payment failure during upgrade	Original plan remains active
Invoice Generation Failure	Partial line item creation	No incomplete invoices created
Usage Event Processing	Batch processing with failures	Either all events processed or none
Concurrent Plan Changes	Multiple simultaneous changes	Serializable execution order

Concurrency testing validates the system's behavior under concurrent load. These tests simulate multiple users performing subscription operations simultaneously to ensure that optimistic locking prevents data corruption and that retry logic handles version conflicts correctly.

Cross-Component Data Flow Testing

Cross-component data flow testing validates that events and data propagate correctly through the system's event-driven architecture. These tests verify that state changes in one component trigger appropriate actions in dependent components.

The test framework publishes domain events and verifies that all registered event handlers execute correctly and that the system reaches a consistent final state. This validation ensures that the loosely-coupled architecture maintains strong consistency guarantees.

Data Flow	Event Triggers	Expected Propagation
Plan Change Event	Proration calculation completion	Invoice line item creation
Usage Event Submission	Batch aggregation completion	Overage charge calculation
Payment Success Event	Gateway webhook processing	Subscription activation
Dunning Escalation	Payment retry failure	Customer notification trigger

Milestone Validation Checkpoints

Each project milestone requires specific validation checkpoints that verify the implemented functionality meets the acceptance criteria. These checkpoints combine automated testing with manual verification procedures to ensure comprehensive validation.

Milestone 1: Plans & Pricing Validation

Plan Management Validation focuses on the flexible pricing plan definition and feature entitlement systems. The validation process ensures that plans support multiple pricing models, handle currency conversion correctly, and maintain proper versioning for existing customers.

Validation Area	Automated Tests	Manual Verification
Plan Definition	Schema validation, pricing model tests	Admin interface plan creation
Currency Support	Multi-currency pricing calculations	Currency display and conversion
Feature Entitlements	Access control logic testing	Feature flag behavior verification
Plan Versioning	Grandfathering logic validation	Existing customer protection testing

Pricing Model Validation requires mathematical verification across all supported pricing structures. The validation process generates test scenarios for flat-rate, tiered, and volume-based pricing to ensure accurate charge calculations.

The checkpoint procedure involves creating test plans with known pricing parameters, then validating that charge calculations produce expected results across various usage quantities and time periods. Property-based testing generates additional scenarios to verify mathematical invariants.

Acceptance Criteria Validation:

- Plan Definition Schema:** Create plans with different intervals, currencies, and feature lists through the management API
- Pricing Model Support:** Validate flat-rate, per-unit, and tiered pricing calculations with known test cases
- Feature Entitlement Matrix:** Verify that plan tiers correctly enable/disable feature flags
- Plan Versioning:** Confirm that plan updates create new versions while preserving existing subscriber access

Milestone 2: Subscription Lifecycle Validation

Subscription State Machine Validation verifies that subscription state transitions follow business rules and that invalid transitions are properly rejected. The validation process tests each valid state transition and confirms that appropriate actions are triggered.

State Transition	Validation Test	Expected Behavior
Trial to Active	Payment method validation	Automatic conversion on trial expiry
Active to Past Due	Payment failure simulation	Dunning process initiation
Past Due to Unpaid	Retry exhaustion	Service suspension
Active to Cancelled	Customer-initiated cancellation	Service termination scheduling

Renewal Processing Validation ensures that the billing cycle engine correctly generates recurring charges, processes payments, and handles failures appropriately. The validation creates test subscriptions with known billing parameters and verifies invoice generation accuracy.

Dunning Management Validation tests the payment failure recovery system by simulating various payment decline scenarios and verifying that retry attempts follow the configured dunning schedule.

Acceptance Criteria Validation:

1. **Subscription Creation:** Provision new subscriptions with selected plans and payment methods
2. **Plan Change Processing:** Execute upgrades/downgrades with correct effective date logic
3. **Cancellation Handling:** Process cancellations with reason recording and termination scheduling
4. **Pause/Resume Functionality:** Handle subscription pausing with prorated billing adjustments

Milestone 3: Proration & Plan Changes Validation

Proration Calculation Validation focuses on mathematical accuracy of partial billing calculations. The validation process tests proration scenarios across different billing intervals, plan types, and change timing to ensure consistent results.

Proration Scenario	Test Data	Validation Method
Mid-cycle Upgrade	Known plan prices, change date	Mathematical verification of charges
Downgrade Credits	Higher-tier plan, remaining days	Credit calculation accuracy
Quantity Changes	Seat-based plans, usage changes	Per-unit proration validation
Currency Handling	Multi-currency plans	Precision and rounding verification

Credit Balance Management Validation ensures that customer credits are properly tracked, applied automatically to invoices, and maintained across billing cycles. The validation process creates scenarios with various credit sources and verifies correct application order.

Plan Change Orchestration Validation tests the end-to-end plan change process, from customer request through proration calculation to service level adjustment. This validation ensures that complex plan changes maintain system consistency.

Acceptance Criteria Validation:

1. **Proration Accuracy:** Calculate prorated charges for upgrades based on remaining cycle days
2. **Credit Application:** Apply downgrade credits for unused portions of higher-tier plans
3. **Mid-cycle Changes:** Handle plan changes with correct effective dates and amounts
4. **Change Scheduling:** Support immediate vs end-of-cycle plan change options

Milestone 4: Usage-Based Billing Validation

Usage Event Processing Validation verifies that metered usage events are correctly ingested, deduplicated, and aggregated for billing purposes. The validation process submits known usage quantities and verifies accurate aggregation across billing periods.

Usage Validation	Test Scenario	Expected Result
Event Deduplication	Duplicate submission with same idempotency key	Single event counted
Billing Period Aggregation	Events across multiple periods	Correct period assignment
Usage Type Isolation	Multiple metrics per customer	Separate aggregation per metric
Overage Calculation	Usage exceeding plan allowances	Accurate tiered rate application

Overage Billing Validation ensures that usage beyond plan allowances is correctly calculated using tiered pricing structures. The validation creates usage scenarios that span multiple pricing tiers and verifies accurate charge calculation.

Usage Quota Enforcement Validation tests the real-time quota checking system that prevents customers from exceeding plan allowances. The validation verifies that quota checks return accurate current usage and remaining allowances.

Acceptance Criteria Validation:

- Idempotent Event Ingestion:** Submit usage events with deduplication protection
- Billing Period Aggregation:** Aggregate usage across correct billing boundaries
- Tiered Usage Charges:** Calculate charges based on usage tier structures
- Overage Rate Application:** Apply correct per-unit overage rates when usage exceeds allowances

Common Testing Pitfalls

⚠ Pitfall: Currency Precision Testing

Many developers test billing calculations using floating-point arithmetic, which introduces precision errors that compound across many transactions. For example, testing a \$9.99 monthly subscription with a 33% proration might use `9.99 * 0.33` in the test, yielding `3.2967` which doesn't represent any valid currency amount.

The correct approach stores all monetary values as integer cents and performs calculations in integer arithmetic. Test assertions should compare exact cent amounts: `assert proration.amount_cents == 329` rather than floating-point approximations.

⚠ Pitfall: Time Zone Inconsistency in Testing

Billing systems operate across multiple time zones, but tests often use local system time, creating inconsistent results when run in different environments. A test that works in EST might fail in PST due to billing period boundary calculations.

All tests must use UTC timestamps consistently and explicitly test time zone edge cases. Test data should include customers in various time zones to verify that billing anchor dates are calculated correctly regardless of system time zone.

Pitfall: Inadequate Idempotency Testing

Developers often test the "happy path" for idempotent operations but fail to test edge cases like partial failures, network timeouts, and retry scenarios. Testing idempotency with simple duplicate requests misses the complex scenarios that occur in production.

Comprehensive idempotency testing must simulate network failures, database timeouts, and system crashes at various points in request processing. Tests should verify that operations remain idempotent even when internal state changes between retry attempts.

Pitfall: Missing Webhook Timing Tests

Payment webhook testing often assumes immediate delivery and processing, but production webhooks can be delayed, delivered out of order, or arrive after timeouts have occurred. Tests that don't account for webhook timing issues fail to catch state machine edge cases.

Webhook tests must simulate delayed delivery, duplicate webhooks, and out-of-order processing. The test framework should control webhook timing independently from payment processing to verify that the system handles all temporal orderings correctly.

Pitfall: Insufficient Error State Testing

Many billing tests focus on successful operations but provide inadequate coverage of error states and recovery scenarios. Testing only successful plan changes misses the complex rollback logic required when payments fail mid-process.

Error state testing must cover every point where operations can fail and verify that the system maintains consistency. This includes testing partial failures, timeout scenarios, and concurrent modification conflicts that can occur in production environments.

Implementation Guidance

The testing implementation provides comprehensive validation tools for all billing system components. The test suite combines mathematical precision testing for calculations with robust integration testing for component interactions.

Technology Recommendations

Test Category	Simple Option	Advanced Option
Unit Testing Framework	<code>pytest</code> with fixtures	<code>pytest</code> with <code>hypothesis</code> for property testing
Database Testing	SQLite in-memory database	PostgreSQL test containers with transaction rollback
HTTP Testing	<code>requests</code> with mock responses	<code>wiremock</code> for full HTTP simulation
Property-Based Testing	Manual test case generation	<code>hypothesis</code> for automated test generation
Time Testing	Fixed datetime mocking	<code>freezegun</code> for temporal scenario control
Currency Testing	Manual decimal calculations	<code>babel</code> for multi-currency test data

Recommended Test Structure

```
tests/
  unit/
    billing/
      test_proration_calculator.py      ← Mathematical calculation tests
      test_usage_aggregator.py         ← Usage event processing tests
      test_pricing_models.py          ← Pricing calculation tests
      test_feature_entitlements.py    ← Access control tests
    subscription/
      test.lifecycle_manager.py        ← State machine tests
      test.renewal_processor.py       ← Billing cycle tests
      test.dunning_engine.py         ← Payment failure recovery tests
    integration/
      test.billing_workflows.py       ← End-to-end billing processes
      test.payment_gateway.py        ← Payment integration tests
      test.webhook_processing.py     ← Asynchronous event tests
    fixtures/
      billing_test_data.py           ← Test data generation utilities
      payment_gateway_mock.py        ← Payment simulation infrastructure
      database_fixtures.py          ← Database state setup utilities
    milestone_validation/
      test_milestone_1_plans.py      ← Plan management validation
      test_milestone_2_lifecycle.py   ← Subscription lifecycle validation
      test_milestone_3_proration.py  ← Plan change validation
      test_milestone_4_usage.py      ← Usage-based billing validation
```

Core Testing Infrastructure

```
# fixtures/billing_test_data.py - Test data generation utilities          PYTHON

from decimal import Decimal

from datetime import datetime, timedelta

from typing import Dict, List, Optional

import uuid

from dataclasses import dataclass


@dataclass

class TestScenario:

    """Represents a billing test scenario with known inputs and expected outputs."""

    name: str

    setup_data: Dict

    expected_result: Dict

    validation_rules: List[str]

class BillingTestDataGenerator:

    """Generates realistic test data for billing calculations."""

    def create_test_plan(

        self,

        plan_type: str = "flat_rate",

        base_price_cents: int = 999,

        currency: str = "USD",

        billing_interval: str = "month"

    ) -> Dict:

        """Create test plan with specified parameters."""

        # TODO 1: Generate plan_id and set basic plan attributes
```

```
# TODO 2: Create pricing_tiers based on plan_type parameter

# TODO 3: Define feature_entitlements for test scenarios

# TODO 4: Set billing_interval and currency_code

# TODO 5: Return complete plan dictionary for test usage


def create_test_customer(

    self,
    credit_balance_cents: int = 0,
    timezone: str = "UTC"
) -> Dict:

    """Create test customer with specified credit balance."""

    # TODO 1: Generate customer_id and basic customer data

    # TODO 2: Set credit_balance_cents and timezone

    # TODO 3: Create default payment method for testing

    # TODO 4: Return customer dictionary with payment method


def create_billing_scenario(

    self,
    scenario_type: str,
    plan_data: Dict,
    customer_data: Dict
) -> TestScenario:

    """Generate complete test scenario with expected results."""

    # TODO 1: Create subscription linking customer to plan

    # TODO 2: Generate usage events based on scenario_type

    # TODO 3: Calculate expected proration or usage charges

    # TODO 4: Define validation rules for test assertions
```

```
# TODO 5: Return TestScenario with all test data and expectations

class ProrationTestGenerator:

    """Specialized test data for proration calculations."""

    def generate_plan_change_scenarios(self) -> List[TestScenario]:
        """Generate comprehensive plan change test cases."""

        # TODO 1: Create upgrade scenarios with various timing
        # TODO 2: Create downgrade scenarios with credit calculations
        # TODO 3: Create quantity change scenarios for seat-based plans
        # TODO 4: Add edge cases for billing period boundaries
        # TODO 5: Include multi-currency scenarios
        # TODO 6: Return list of all generated test scenarios

    def generate_calendar_edge_cases(self) -> List[TestScenario]:
        """Generate calendar edge case scenarios for proration."""

        # TODO 1: Create leap year scenarios (Feb 29 billing anchor)
        # TODO 2: Create month overflow scenarios (Jan 31 + 1 month)
        # TODO 3: Create timezone transition scenarios (DST changes)
        # TODO 4: Create end-of-month billing scenarios
        # TODO 5: Return comprehensive calendar edge case scenarios
```

Property-Based Testing Framework

```
# tests/unit/billing/test_proration_properties.py - Property-based proration testing PYTHON

from hypothesis import given, strategies as st, assume

from decimal import Decimal

from datetime import date, timedelta

import pytest


class ProrationPropertyTests:

    """"Property-based tests for proration mathematical invariants.""""

    @given(
        old_price=st.integers(min_value=100, max_value=100000), # $1-$1000 in cents
        new_price=st.integers(min_value=100, max_value=100000),
        change_date=st.dates(min_value=date(2024, 1, 1), max_value=date(2024, 12, 31)),
        billing_start=st.dates(min_value=date(2024, 1, 1), max_value=date(2024, 6, 30)),
    )

    def test_proration_symmetry_property(
            self, old_price: int, new_price: int, change_date: date, billing_start: date
    ):

        """"Verify that upgrade followed by immediate downgrade nets to zero.""""

        # Ensure change_date is within billing period
        billing_end = billing_start + timedelta(days=30)

        assume(billing_start <= change_date <= billing_end)

        # TODO 1: Calculate proration for upgrade from old_price to new_price

        # TODO 2: Calculate proration for immediate downgrade back to old_price

        # TODO 3: Verify that net proration amount is zero (within rounding tolerance)
```

```
# TODO 4: Verify that both calculations use same time factor

# Hint: Use ProrationCalculator.calculate_plan_change_proration for both
calculations

@given(
    base_price=st.integers(min_value=500, max_value=10000),
    period_days=st.integers(min_value=28, max_value=31),
    change_day=st.integers(min_value=1, max_value=31),
)

def test_proration_monotonicity(
    self, base_price: int, period_days: int, change_day: int
):
    """Verify that later change dates result in smaller proration amounts."""

    assume(change_day <= period_days)

    # TODO 1: Calculate proration for change on change_day
    # TODO 2: Calculate proration for change one day later (if within period)
    # TODO 3: Verify that later change has smaller absolute proration amount
    # TODO 4: Handle edge case where change_day is last day of period

    # Hint: Proration amount should decrease as change_date approaches period_end
```

Integration Test Infrastructure

```
# tests/integration/test_billing_workflows.py - End-to-end billing workflow testing    PYTHON

import pytest

from datetime import datetime, timedelta

from unittest.mock import Mock, patch

import json


class BillingWorkflowIntegrationTests:

    """Integration tests for complete billing workflows."""

    @pytest.fixture
    def billing_test_environment(self, db_session):
        """Set up complete test environment with all components."""
        # TODO 1: Create test database with all billing tables
        # TODO 2: Initialize mock payment gateway with test responses
        # TODO 3: Set up event bus with test event handlers
        # TODO 4: Create test customers, plans, and subscriptions
        # TODO 5: Return configured test environment object

    def test_complete_subscription_lifecycle(self, billing_test_environment):
        """Test complete customer journey from signup to cancellation."""
        env = billing_test_environment

        # TODO 1: Create customer with payment method
        # TODO 2: Create subscription with trial period
        # TODO 3: Process trial expiration and first payment
        # TODO 4: Generate usage events and process monthly billing
```

```
# TODO 5: Process plan upgrade with proration

# TODO 6: Cancel subscription and verify final billing

# TODO 7: Validate all audit events and state transitions


# Validation points

# TODO 8: Verify subscription state progression is correct

# TODO 9: Verify all invoices have accurate line items

# TODO 10: Verify payment processing completed successfully

# TODO 11: Verify audit trail captures all operations


def test_payment_failure_recovery(self, billing_test_environment):

    """Test complete dunning process for failed payments."""

    env = billing_test_environment


    # TODO 1: Create active subscription with invalid payment method

    # TODO 2: Trigger billing cycle and simulate payment failure

    # TODO 3: Verify dunning process initiates correctly

    # TODO 4: Process multiple retry attempts with continued failures

    # TODO 5: Verify subscription transitions to unpaid status

    # TODO 6: Test payment method update and recovery

    # TODO 7: Verify subscription reactivates after successful payment
```

Milestone Validation Scripts

```
# tests/milestone_validation/test_milestone_1_plans.py - Plan management milestone validation          PYTHON

import pytest

from decimal import Decimal


class Milestone1ValidationTests:

    """Comprehensive validation for Milestone 1: Plans & Pricing."""

    def test_plan_definition_schema(self, plan_manager):

        """Validate plan definition supports required schema elements."""

        # Acceptance Criteria: Define subscription plans with tiers, intervals, and pricing

        # TODO 1: Create flat-rate plan with monthly billing interval

        # TODO 2: Create tiered plan with multiple pricing tiers

        # TODO 3: Create per-seat plan with quantity-based pricing

        # TODO 4: Verify all plans store correctly with complete schema

        # TODO 5: Test plan retrieval and validation of all fields

        # Expected behaviors:

        # - Plan creation returns valid plan_id

        # - All pricing tiers store with correct boundaries

        # - Feature entitlements associate correctly with plans

        # - Currency and interval settings persist accurately

    def test_pricing_model_calculations(self, plan_manager):

        """Validate pricing model charge calculations."""

        # Acceptance Criteria: Support multiple pricing models (flat, tiered)

        # TODO 1: Test flat-rate pricing calculation for various quantities
```

```
# TODO 2: Test tiered pricing with usage spanning multiple tiers

# TODO 3: Test volume pricing with rate changes at thresholds

# TODO 4: Verify currency conversion maintains precision

# TODO 5: Test edge cases at tier boundaries

# Mathematical validation:

# - Flat-rate: charge = base_price * quantity

# - Tiered: charge = sum(tier_rate * tier_quantity for each tier)

# - Volume: charge = total_quantity * applicable_rate

def test_plan_versioning_protection(self, plan_manager):

    """Validate plan versioning protects existing customers."""

    # Acceptance Criteria: Handle plan versioning for existing subscribers

    # TODO 1: Create plan and subscribe test customer

    # TODO 2: Update plan pricing and create new version

    # TODO 3: Verify existing customer retains original pricing

    # TODO 4: Verify new customers receive updated pricing

    # TODO 5: Test plan deprecation and grandfather protection

    # Validation checkpoints:

    print("v Plan creation and versioning")

    print("v Existing customer price protection")

    print("v New customer updated pricing")

    print("v Feature entitlement inheritance")
```

Debugging Test Utilities

Test Failure Symptom	Likely Cause	Diagnostic Command	Resolution
Proration calculation mismatch	Floating point precision error	Check test uses integer cents	Convert all amounts to integer cents
Webhook test timing out	Mock gateway not configured	Verify mock responses set up	Initialize payment gateway mock properly
Database constraint violation	Test data conflicts with existing	Use transaction rollback fixtures	Isolate tests with database transactions
Property test failure	Edge case not handled	Run test with <code>-- hypothesis-verbose</code>	Add assume() statements for valid inputs
Integration test flakiness	Race condition in async processing	Add explicit event synchronization	Use event bus sync points in tests

Milestone Checkpoint Validation

Milestone 1 Checkpoint:

```
# Run plan management validation                                         BASH
python -m pytest tests/milestone_validation/test_milestone_1_plans.py -v

# Expected output:

# ✓ test_plan_definition_schema - PASSED
# ✓ test_pricing_model_calculations - PASSED
# ✓ test_plan_versioning_protection - PASSED
# ✓ test_feature_entitlements - PASSED

# Manual verification:

# 1. Create plan via API: POST /api/plans with plan data
# 2. Verify response includes plan_id and all specified fields
# 3. Retrieve plan: GET /api/plans/{plan_id}
# 4. Confirm pricing tiers and features are correct
```

Milestone 2 Checkpoint:

```
# Run subscription lifecycle validation                                BASH
python -m pytest tests/milestone_validation/test_milestone_2_lifecycle.py -v

# Expected behaviors to verify:
# 1. Subscription creation provisions customer access
# 2. Renewal processing generates accurate invoices
# 3. Cancellation respects end-of-period vs immediate options
# 4. State transitions follow business rules
```

Milestone 3 Checkpoint:

```
# Run proration calculation validation                                BASH
python -m pytest tests/milestone_validation/test_milestone_3_proration.py -v

# Mathematical verification:
# 1. Mid-cycle upgrade: charge = (new_price - old_price) * time_factor
# 2. Downgrade credit: credit = (old_price - new_price) * time_factor
# 3. Time factor: remaining_days / total_period_days
# 4. Currency precision: all calculations in integer cents
```

Milestone 4 Checkpoint:

```
# Run usage-based billing validation                                BASH
python -m pytest tests/milestone_validation/test_milestone_4_usage.py -v

# Usage processing verification:
# 1. Submit usage events with idempotency keys
# 2. Verify aggregation across billing periods
# 3. Calculate overage charges for usage beyond allowances
# 4. Test quota enforcement and limit checking
```

The testing strategy provides comprehensive validation across all billing system components, ensuring mathematical accuracy, integration reliability, and milestone completion verification. The property-based testing approach catches edge cases that manual test writing might miss, while the milestone validation checkpoints provide clear success criteria for each development phase.

Debugging Guide

Milestone(s): All milestones - provides diagnostic techniques and troubleshooting approaches essential for identifying and resolving issues across plan management (Milestone 1), subscription lifecycle (Milestone 2), proration calculations (Milestone 3), and usage-based billing (Milestone 4)

Mental Model: Detective Work for Financial Systems

Think of debugging a billing system like being a financial detective investigating discrepancies in a complex accounting operation. Just as a detective follows the money trail through bank records, receipts, and transaction logs, billing system debugging requires tracing data flow through multiple components, examining audit trails, and understanding the sequence of events that led to incorrect calculations or inconsistent states.

Unlike debugging typical web applications where the worst outcome might be a 500 error, billing system bugs directly impact revenue and customer trust. A proration calculation error could overcharge thousands of customers, while a webhook processing failure might leave subscriptions in inconsistent states. The debugging approach must be systematic, thorough, and focused on financial accuracy and data consistency.

The debugging process mirrors forensic accounting: start with the symptom (incorrect invoice amount, stuck subscription), trace backwards through the audit trail (events, state changes, calculations), identify the root cause (rounding error, race condition, webhook duplication), and implement fixes that prevent recurrence. Every financial operation must be explainable and recoverable.

Billing Calculation Issues

Billing calculation debugging requires understanding the mathematical precision, currency handling, and complex interaction between base charges, usage fees, proration, and credits. These issues often manifest as subtle discrepancies that compound over time or edge cases that only occur under specific conditions.

Currency Precision and Rounding Errors

Currency precision errors are among the most common and dangerous billing calculation issues. These problems typically stem from mixing floating-point arithmetic with financial calculations or incorrect rounding when converting between different currency representations.

Common Currency Precision Problems:

Problem Type	Symptom	Root Cause	Detection Method
Floating Point Drift	Invoice totals off by 1-2 cents	Using <code>float</code> instead of integer cents	Compare sum of line items vs total amount
Rounding Inconsistency	Line item totals don't match invoice total	Different rounding at each calculation step	Recalculate invoice from scratch and compare
Currency Conversion Precision	Multi-currency invoices with tiny discrepancies	Exchange rate precision loss	Check conversion rate precision and intermediate values
Aggregate Rounding	Usage charges incorrect for high quantities	Rounding per-unit vs rounding total	Compare unit-by-unit vs bulk calculation methods

The fundamental principle for currency debugging is that all monetary values must be stored and calculated using the smallest currency unit (cents for USD). Any conversion to decimal representation should happen only at the display layer, never during calculations.

Proration Calculation Debugging:

Proration errors are particularly insidious because they involve both time-based calculations and currency precision. The debugging approach requires validating both the time fraction calculation and the monetary arithmetic.

Proration Issue	Diagnostic Steps	Validation Method
Incorrect Time Fraction	Log change date, period start, period end	Calculate days manually: <code>(period_end - change_date) / (period_end - period_start)</code>
Asymmetric Proration	Upgrade charge doesn't match downgrade credit	Test upgrade followed by immediate downgrade - net should be zero
Calendar Edge Cases	Wrong proration on month boundaries	Test month-end changes, leap year scenarios, timezone boundaries
Multiple Plan Changes	Cumulative proration errors	Sum all proration amounts for billing period - should equal net plan difference

The proration debugging process should always start with validating the time calculations before examining monetary arithmetic. A common mistake is debugging currency precision when the actual issue is incorrect date handling or timezone problems.

Usage Aggregation and Tiered Pricing Issues:

Usage-based billing introduces additional complexity because charges depend on both quantity calculations and tier boundary logic. Debugging requires understanding the aggregation methodology and tier calculation algorithms.

Usage Billing Problem	Investigation Approach	Key Validation
Incorrect Usage Totals	Trace individual events to final aggregation	Manual sum of events vs aggregated quantity
Wrong Tier Application	Examine tier boundary conditions	Test quantities at tier boundaries (999, 1000, 1001)
Duplicate Event Charging	Check idempotency key handling	Search for duplicate <code>idempotency_key</code> values
Missing Overage Charges	Validate allowance vs actual usage	Compare plan allowance against aggregated usage
Cross-Period Event Attribution	Check event timestamps vs billing periods	Verify events assigned to correct billing cycle

Usage debugging often requires examining the raw event data and manually recalculating aggregations. The `UsageEvent` table should be treated as the source of truth, and all aggregated values should be derivable from this base data.

Credit Balance and Application Errors

Credit balance issues occur when customer credits aren't applied correctly to invoices, or when credit calculations don't match the corresponding charges. These problems often involve incorrect sequencing of credit operations or failure to handle partial credit applications.

Credit Balance Debugging Matrix:

Credit Issue Type	Symptoms	Diagnostic Query	Expected Result
Credits Not Applied	Invoice amount ignores available credit	<code>SELECT * FROM customer WHERE customer_id = ? AND credit_balance_cents > 0</code>	Credit balance should decrease after invoice
Partial Credit Application	Credit partially consumed but amount wrong	Trace credit consumption in audit logs	Credit reduction should equal amount applied
Negative Credit Balance	Customer credit balance below zero	Check for concurrent credit consumption	Credit operations should be atomic
Orphaned Credits	Credits exist but never applied to invoices	Find credits without corresponding invoice line items	All credits should have application records

The credit debugging process requires examining both the current credit balance and the complete history of credit applications through the audit trail. Every credit operation should be traceable through the `AuditEvent` records.

State Consistency Problems

State consistency issues in billing systems typically involve subscription states becoming misaligned with payment states, invoice states, or usage tracking states. These problems often result from partial failures in distributed operations or race conditions between concurrent processes.

Subscription State Inconsistencies

Subscription state debugging requires understanding the valid state transitions and identifying how the subscription reached an invalid or unexpected state. The subscription state machine provides the framework for validating state consistency.

Subscription State Validation Checklist:

Current State	Required Conditions	Inconsistency Indicators	Recovery Actions
<code>active</code>	Valid payment method, current invoice paid	No payment method or unpaid invoice	Check payment status and payment method validity
<code>past_due</code>	Active dunning process, unpaid invoice exists	No dunning attempts or no unpaid invoices	Verify dunning engine status and invoice generation
<code>cancelled</code>	Cancellation event exists, no future invoices	Active invoices scheduled for future	Cancel future invoices and verify cancellation timestamp
<code>paused</code>	Pause event exists, billing suspended	Recent invoices or payments	Check pause effective date and billing suspension

The key to subscription state debugging is examining the sequence of `SubscriptionEvent` records that led to the current state. Every state transition should correspond to a specific event with proper authorization and business justification.

Concurrent Modification Detection:

Race conditions in subscription updates can lead to lost state changes or invalid state transitions. The optimistic locking mechanism using version numbers provides protection against concurrent modifications.

Concurrency Problem	Detection Method	Prevention Strategy
Lost State Updates	Version mismatch exceptions in logs	Retry with fresh entity load
Invalid State Transitions	State machine violations	Validate transitions before persisting
Duplicate Event Processing	Multiple events with same idempotency key	Check idempotency key uniqueness
Timeline Ordering Issues	Events processed out of sequence	Use event timestamp ordering for replay

When debugging concurrency issues, examine the `WorkflowExecution` records to understand which operations were running simultaneously and whether any workflow steps failed due to conflicts.

Invoice and Payment State Alignment

Invoice states must remain consistent with payment states and subscription states. Misalignment often occurs when webhook processing fails or when manual interventions bypass the normal workflow.

Invoice-Payment Consistency Matrix:

Invoice Status	Expected Payment Status	Inconsistency Pattern	Resolution Steps
<code>draft</code>	No payments exist	Payments exist for draft invoice	Investigate manual payment creation
<code>open</code>	<code>pending</code> or no payment	<code>succeeded</code> payment but invoice still open	Process payment success webhook
<code>paid</code>	<code>succeeded</code> payment	No successful payment found	Check for manual payment recording
<code>past_due</code>	<code>failed</code> payment	<code>succeeded</code> payment	Investigate webhook processing delay

Invoice state debugging requires correlating invoice records with payment records using the `invoice_id` foreign key. Every paid invoice should have exactly one successful payment, and every successful payment should correspond to a paid invoice.

Payment Gateway Synchronization Issues:

Billing system state can become inconsistent with payment gateway state when webhooks fail to process or when manual operations occur in the gateway without corresponding system updates.

Synchronization Issue	Detection Strategy	Reconciliation Approach
Missing Webhook Processing	Gateway payment exists, no system payment	Compare gateway transactions with system payments
Duplicate Payment Recording	Multiple system payments for single gateway charge	Search for payments with same <code>gateway_transaction_id</code>
Amount Mismatches	System payment amount differs from gateway	Compare <code>amount_cents</code> with gateway charge amount
Status Lag	Gateway shows success, system shows pending	Check webhook event processing timestamps

Payment gateway debugging often requires accessing the gateway's dashboard or API to compare transaction states with internal system records. The `WebhookEvent` table provides an audit trail of all webhook processing attempts.

Webhook and Integration Debugging

Webhook processing failures can leave the billing system in inconsistent states because external payment events aren't properly reflected in system state. Debugging webhook issues requires understanding both the technical integration mechanisms and the business implications of processing delays or failures.

Webhook Processing Reliability

Webhook reliability debugging focuses on ensuring that payment gateway events are consistently processed and applied to the appropriate billing entities. This involves examining the webhook ingestion pipeline, idempotency protection, and error handling.

Webhook Processing Diagnostic Flow:

Processing Stage	Success Indicators	Failure Symptoms	Debug Actions
Webhook Reception	Request logged with valid signature	400/500 errors in webhook endpoint	Check webhook URL configuration and signature validation
Event Parsing	<code>WebhookEvent</code> record created	JSON parsing errors in logs	Validate webhook payload structure against gateway documentation
Idempotency Check	Duplicate events ignored gracefully	Same event processed multiple times	Verify <code>idempotency_key</code> uniqueness enforcement
Business Logic Application	Entity states updated correctly	State inconsistencies after webhook	Trace event processing through business logic

The webhook debugging process should start by confirming that webhooks are being received by the system. Check the webhook endpoint logs for incoming requests and verify that the payment gateway is configured with the correct URL and credentials.

Webhook Event Deduplication Issues:

Payment gateways may send duplicate webhooks due to network retries or internal processing issues. The idempotency protection mechanism must correctly identify and ignore duplicate events without losing legitimate updates.

Deduplication Problem	Identification Method	Root Cause Analysis
Duplicate Processing	Same payment updated multiple times	Check <code>idempotency_key</code> uniqueness in <code>WebhookEvent</code> table
False Duplicate Detection	Legitimate events ignored	Examine idempotency key generation logic
Race Condition in Deduplication	Intermittent duplicate processing	Check concurrent webhook processing timing
Cross-Event Type Conflicts	Different event types sharing idempotency keys	Validate idempotency key scoping strategy

Effective webhook deduplication debugging requires examining both the incoming webhook payloads and the generated idempotency keys. The idempotency key should uniquely identify each business event, not just each HTTP request.

Payment Method and Subscription Synchronization

Payment method updates in the gateway must be reflected in subscription billing configurations to ensure future payments succeed. Debugging these synchronization issues involves tracing payment method lifecycle events through both systems.

Payment Method Sync Debugging:

Sync Issue	System Impact	Debugging Approach	Resolution Strategy
Outdated Payment Method	Recurring payments fail	Compare system vs gateway payment method status	Refresh payment method from gateway API
Missing Payment Method Updates	Card expiration not reflected	Check webhook processing for payment method events	Implement payment method webhook handlers
Customer Payment Method Misalignment	Payments charged to wrong method	Validate customer-payment method associations	Reconcile customer records between systems
Default Payment Method Conflicts	Multiple methods marked as default	Query payment methods for customer	Implement default payment method enforcement

Payment method debugging often requires cross-referencing data between the billing system and payment gateway. The `PaymentMethod` table should maintain accurate references to gateway payment method tokens and their current status.

Integration Circuit Breaker and Retry Logic:

When external integrations fail repeatedly, circuit breakers prevent cascading failures while retry logic attempts to recover from transient issues. Debugging these mechanisms helps identify when external services are degraded and how the system responds.

Integration Health Issue	Circuit Breaker State	Debug Information	Recovery Actions
Gateway API Timeouts	Circuit opening frequently	Check response time metrics and failure rates	Increase timeout values or investigate gateway performance
Authentication Failures	Circuit stuck open	Examine API key validity and rotation	Refresh authentication credentials
Rate Limiting	Requests failing with 429 errors	Check request volume and rate limit headers	Implement exponential backoff with jitter
Webhook Endpoint Unreachable	Gateway shows delivery failures	Verify webhook endpoint availability and SSL certificate	Fix infrastructure issues and re-register webhook URL

Circuit breaker debugging requires examining both the current circuit state and the historical failure patterns that triggered the protection mechanism. The `CircuitBreaker` component should provide visibility into failure thresholds and recovery timing.

Implementation Guidance

The debugging implementation focuses on providing comprehensive diagnostic tools, structured logging, and automated consistency checks that help identify and resolve billing system issues efficiently.

Technology Recommendations:

Debugging Component	Simple Option	Advanced Option
Logging Framework	Python <code>logging</code> with structured JSON	ELK Stack (Elasticsearch, Logstash, Kibana)
Metrics Collection	Python <code>prometheus_client</code>	Datadog or New Relic APM
Database Debugging	Direct SQL queries with psql/mysql	Database query analysis tools
API Testing	<code>curl</code> and <code>pytest</code>	Postman collections with test automation
Error Tracking	Python <code>traceback</code> with file output	Sentry error tracking and alerting

Recommended File Structure for Debugging Tools:

```
project-root/
  src/
    billing/
      debug/
        __init__.py
        calculation_validator.py      ← billing calculation verification
        state_consistency_checker.py ← subscription state validation
        webhook_debugger.py          ← webhook processing diagnostics
        audit_trail_analyzer.py     ← audit event investigation tools
        billing_reconciliation.py   ← system vs gateway comparison
        test_data_generator.py       ← synthetic data for testing scenarios
      monitoring/
        __init__.py
        billing_metrics.py           ← prometheus metrics collection
        health_checks.py            ← system health validation
        circuit_breaker_monitor.py  ← integration health monitoring
    tools/
      debug_billing_issue.py      ← command-line debugging utility
      reconcile_payments.py       ← payment gateway synchronization
      validate_billing_period.py  ← billing cycle validation script
  tests/
    debug_tools/
      test_calculation_validator.py
      test_state_consistency.py
      test_webhook_debugger.py
```

Infrastructure Starter Code - Billing Calculation Validator:

```
import logging

from decimal import Decimal, ROUND_HALF_UP

from typing import Dict, List, Optional, Any

from dataclasses import dataclass

from enum import Enum


class ValidationError(Enum):

    CURRENCY_PRECISION = "currency_precision"

    PRORATION_ASYMMETRY = "proration_asymmetry"

    USAGE_AGGREGATION = "usage_aggregation"

    CREDIT_APPLICATION = "credit_application"

    TIER_CALCULATION = "tier_calculation"

    @dataclass

    class ValidationResult:

        is_valid: bool

        error_type: Optional[ValidationError]

        expected_value: Any

        actual_value: Any

        details: Dict[str, Any]

    class BillingCalculationValidator:

        """"

        Validates billing calculations for correctness and consistency.

        Provides detailed diagnostics for calculation errors.

        """

    def __init__(self, decimal_precision: Decimal = Decimal('0.01')):
```

PYTHON

```
self.decimal_precision = decimal_precision

self.logger = logging.getLogger(__name__)

def validate_currency_precision(self, money_amount: 'Money') -> ValidationResult:
    """
    Validates that monetary amounts maintain proper precision.

    Checks for floating-point drift and rounding consistency.

    """
    # Convert to decimal and back to check for precision loss
    decimal_amount = money_amount.to_decimal()

    reconstructed = Money.from_decimal(decimal_amount, money_amount.currency_code)

    if reconstructed.amount_cents != money_amount.amount_cents:
        return ValidationResult(
            is_valid=False,
            error_type=ValidationError.CURRENCY_PRECISION,
            expected_value=money_amount.amount_cents,
            actual_value=reconstructed.amount_cents,
            details={
                'original_cents': money_amount.amount_cents,
                'decimal_representation': str(decimal_amount),
                'reconstructed_cents': reconstructed.amount_cents,
                'precision_loss': money_amount.amount_cents -
                    reconstructed.amount_cents
            }
        )
    
```

```
        return ValidationResult(is_valid=True, error_type=None, expected_value=None,
actual_value=None, details={})

def validate_proration_symmetry(self, old_plan: 'Plan',
                                 change_date: datetime, billing_period_start: datetime,
                                 billing_period_end: datetime) -> ValidationResult:
    """
    Validates that upgrade followed by immediate downgrade results in zero net charge.

    Tests proration calculation symmetry and consistency.
    """

    # Calculate upgrade proration
    upgrade_result = ProrationCalculator().calculate_plan_change_proration(
        old_plan, new_plan, change_date, billing_period_start, billing_period_end
    )

    # Calculate immediate downgrade proration
    downgrade_result = ProrationCalculator().calculate_plan_change_proration(
        new_plan, old_plan, change_date, billing_period_start, billing_period_end
    )

    # Net amount should be zero (within precision tolerance)
    net_charge = upgrade_result.net_amount.amount_cents +
    downgrade_result.net_amount.amount_cents

    tolerance_cents = 1  # Allow 1 cent tolerance for rounding

    if abs(net_charge) > tolerance_cents:
        return ValidationResult(
            is_valid=False,
```

```

        error_type=ValidationError.PRORATION_ASYMMETRY,
        expected_value=0,
        actual_value=net_charge,
        details={

            'upgrade_net_cents': upgrade_result.net_amount.amount_cents,
            'downgrade_net_cents': downgrade_result.net_amount.amount_cents,
            'total_net_cents': net_charge,
            'tolerance_cents': tolerance_cents,
            'upgrade_details': upgrade_result.calculation_details,
            'downgrade_details': downgrade_result.calculation_details
        }
    )

    return ValidationResult(is_valid=True, error_type=None, expected_value=None,
actual_value=None, details={})
}

class StateConsistencyChecker:

    """
    Validates subscription and billing state consistency.

    Identifies state machine violations and data synchronization issues.
    """

    def __init__(self, database_manager: 'DatabaseManager'):

        self.db = database_manager

        self.logger = logging.getLogger(__name__)

    def check_subscription_state_consistency(self, subscription_id: str) ->
List[ValidationResult]:

```

```
"""

Comprehensive subscription state validation.

Checks state machine compliance and business rule adherence.

"""

results = []

with self.db.transaction() as session:

    subscription =
session.query(Subscription).filter_by(subscription_id=subscription_id).first()

    if not subscription:

        return [ValidationResult(
            is_valid=False,
            error_type=ValidationError.CONSISTENCY,
            expected_value="subscription exists",
            actual_value="subscription not found",
            details={'subscription_id': subscription_id}
        )]

        # Validate state-specific conditions

        results.extend(self._validate_active_subscription_requirements(subscription,
session))

        results.extend(self._validate_past_due_subscription_requirements(subscription,
session))

        results.extend(self._validate_cancelled_subscription_requirements(subscription,
session))

return results
```



```
    Diagnoses webhook processing issues and payment gateway synchronization problems.

    Provides detailed analysis of webhook event flow and processing status.

    """
    """
```



```
def __init__(self, database_manager: 'DatabaseManager'):

    self.db = database_manager

    self.logger = logging.getLogger(__name__)
```



```
def analyze_webhook_processing(self, gateway_event_id: str) -> Dict[str, Any]:
    """
        Comprehensive webhook processing analysis.

        Traces event from gateway through system processing.

    """
    analysis = {

        'gateway_event_id': gateway_event_id,

        'processing_status': 'unknown',

        'processing_attempts': [],

        'business_impact': {},

        'recommendations': []
    }

    with self.db.transaction() as session:

        # Find webhook event records

        webhook_events = session.query(WebhookEvent).filter_by(
            gateway_event_id=gateway_event_id
        ).order_by(WebhookEvent.created_at).all()
```

```
if not webhook_events:

    analysis['processing_status'] = 'not_received'

    analysis['recommendations'].append('Verify webhook URL configuration in
payment gateway')

    return analysis


# Analyze processing attempts

for event in webhook_events:

    attempt_info = {

        'event_id': event.event_id,

        'received_at': event.created_at.isoformat(),

        'processed_at': event.processed_at.isoformat() if event.processed_at
else None,

        'idempotency_key': event.idempotency_key,

        'event_type': event.event_type

    }

    analysis['processing_attempts'].append(attempt_info)


# Check for successful processing

processed_events = [e for e in webhook_events if e.processed_at is not None]

if processed_events:

    analysis['processing_status'] = 'processed'

    analysis['business_impact'] =
self._analyze_business_impact(processed_events[-1], session)

else:

    analysis['processing_status'] = 'failed'

    analysis['recommendations'].extend([
        'Check application logs for processing errors',
        'Review database logs for any errors or anomalies during processing'
    ])
```

```
'Verify webhook payload structure matches expected format',  
'Confirm database connectivity during processing window'  
])  
  
return analysis
```

Core Logic Skeleton - Billing Issue Diagnostic Tool:

```
class BillingIssueDiagnostic:
```

PYTHON

```
"""
```

```
Command-line tool for diagnosing billing system issues.
```

```
Provides structured investigation workflow for common problems.
```

```
"""
```

```
def __init__(self, database_manager: 'DatabaseManager'):
```

```
    self.db = database_manager
```

```
    self.validator = BillingCalculationValidator()
```

```
    self.consistency_checker = StateConsistencyChecker(database_manager)
```

```
    self.webhook_debugger = WebhookDebugger(database_manager)
```

```
    self.logger = logging.getLogger(__name__)
```

```
def diagnose_invoice_discrepancy(self, invoice_id: str) -> Dict[str, Any]:
```

```
    """
```

```
    Comprehensive invoice discrepancy diagnosis.
```

```
    Validates all calculations and identifies source of errors.
```

```
    """
```

```
# TODO 1: Load invoice and related entities (subscription, customer, line items)
```

```
# TODO 2: Recalculate invoice total from line items and compare
```

```
# TODO 3: Validate each line item calculation (subscription, usage, proration, credits)
```

```
# TODO 4: Check for currency precision issues in calculations
```

```
# TODO 5: Verify credit application logic and amounts
```

```
# TODO 6: Generate detailed discrepancy report with recommendations
```

```
pass
```

```
def diagnose_subscription_state_issue(self, subscription_id: str) -> Dict[str, Any]:  
    """  
  
    Subscription state diagnostic workflow.  
  
    Identifies invalid states and recommends correction steps.  
  
    """  
  
    # TODO 1: Load subscription and validate current state  
  
    # TODO 2: Trace subscription event history for state transitions  
  
    # TODO 3: Validate state machine compliance for each transition  
  
    # TODO 4: Check for concurrent modification conflicts  
  
    # TODO 5: Verify payment method and invoice alignment with state  
  
    # TODO 6: Generate state correction recommendations  
  
    pass  
  
  
  
def diagnose_payment_processing_issue(self, payment_id: str) -> Dict[str, Any]:  
    """  
  
    Payment processing diagnostic workflow.  
  
    Traces payment through gateway integration and webhook processing.  
  
    """  
  
    # TODO 1: Load payment record and related invoice/subscription  
  
    # TODO 2: Check payment gateway for corresponding transaction  
  
    # TODO 3: Analyze webhook processing for payment events  
  
    # TODO 4: Validate payment state consistency with gateway  
  
    # TODO 5: Check for duplicate payment processing  
  
    # TODO 6: Generate synchronization and recovery recommendations  
  
    pass  
  
  
def main():
```

```
"""Command-line interface for billing system debugging."""

import argparse


parser = argparse.ArgumentParser(description='Billing System Diagnostic Tool')

parser.add_argument('issue_type', choices=['invoice', 'subscription', 'payment',
'webhook'])

parser.add_argument('entity_id', help='ID of entity to investigate')

parser.add_argument('--verbose', action='store_true', help='Enable detailed logging')


args = parser.parse_args()


if args.verbose:

    logging.basicConfig(level=logging.DEBUG)


db_manager = DatabaseManager()

diagnostic = BillingIssueDiagnostic(db_manager)


if args.issue_type == 'invoice':

    result = diagnostic.diagnose_invoice_discrepancy(args.entity_id)

elif args.issue_type == 'subscription':

    result = diagnostic.diagnose_subscription_state_issue(args.entity_id)

elif args.issue_type == 'payment':

    result = diagnostic.diagnose_payment_processing_issue(args.entity_id)

elif args.issue_type == 'webhook':

    result = diagnostic.webhook_debugger.analyze_webhook_processing(args.entity_id)


print(json.dumps(result, indent=2, default=str))
```

```
if __name__ == '__main__':
    main()
```

Milestone Checkpoint - Debugging Tool Validation:

After implementing the debugging tools, validate their effectiveness with these checkpoints:

1. **Currency Precision Validation:** Run `python -m billing.debug.calculation_validator` with test invoices containing known precision issues. The validator should identify floating-point drift and rounding inconsistencies.
2. **State Consistency Checks:** Execute `python tools/debug_billing_issue.py subscription <subscription_id>` for subscriptions in various states. The tool should identify state machine violations and business rule inconsistencies.
3. **Webhook Processing Analysis:** Use `python tools/debug_billing_issue.py webhook <gateway_event_id>` to trace webhook events from payment gateway through system processing. The analysis should show processing status and business impact.
4. **Proration Symmetry Tests:** Execute proration validation tests that perform upgrade followed by immediate downgrade. The net charge should be zero within rounding tolerance.

Common Debugging Pitfalls:

⚠ **Pitfall: Debugging Symptoms Instead of Root Causes** Many developers focus on fixing the immediate symptom (wrong invoice amount) without identifying the underlying issue (currency precision loss in calculations). Always trace the problem back to its source through the audit trail and recalculate values from scratch.

⚠ **Pitfall: Ignoring Timezone Issues in Date Calculations** Billing calculations often involve date arithmetic that can be affected by timezone conversions and daylight saving time changes. Always store timestamps in UTC and convert to local time only for display purposes.

⚠ **Pitfall: Assuming Webhook Processing is Synchronous** Webhook processing is inherently asynchronous, and debugging must account for processing delays, retries, and out-of-order delivery. Check webhook processing timestamps and correlate with business entity state changes.

Future Extensions and Scalability

Milestone(s): All milestones - outlines scalability considerations and future enhancements that build upon the foundation established in plan management (Milestone 1), subscription lifecycle (Milestone 2), proration (Milestone 3), and usage-based billing (Milestone 4)

Mental Model: Evolution of a Growing Business

Think of the subscription billing system as a growing city's infrastructure. Initially, you build a water system that serves a few thousand residents with simple pipes and a single treatment plant. As the city grows, you need to add multiple treatment facilities (multi-tenant architecture), accommodate different types of buildings with varying water needs (advanced pricing models), and build comprehensive monitoring systems to track usage patterns and optimize distribution (analytics and revenue reporting).

The key architectural principle is that your foundational infrastructure must be designed to support this growth without requiring complete reconstruction. Just as a well-planned city leaves room for expansion and upgrades existing utilities rather than replacing them entirely, your billing system's core components must be extensible and composable to support future business requirements.

Advanced Pricing Models

The current system supports flat-rate, tiered, and volume-based pricing through the `PricingModel` enum and `PricingTier` structures. However, modern B2B SaaS businesses often require more sophisticated pricing models that adapt to complex customer requirements and business relationships.

Seat-Based Billing Implementation

Seat-based billing represents a significant extension where charges scale based on the number of active users or licenses within a customer's organization. This model requires tracking user activation and deactivation events throughout the billing cycle, calculating mid-cycle adjustments when users are added or removed, and handling the complexity of different user types with varying pricing.

The extension begins with enhancing the `Plan` entity to support seat-based pricing configurations. This involves adding fields for base seat count, additional seat pricing, and seat type definitions. The system must track seat usage through dedicated usage events that capture user activation, deactivation, and role changes with precise timestamps.

Component Enhancement	Current State	Seat-Based Extension
Plan Definition	Fixed pricing tiers	Dynamic seat multipliers with base counts
Usage Tracking	Generic <code>UsageEvent</code>	Specialized seat activation/deactivation events
Proration Logic	Time-based only	Combination of time and quantity-based proration
Invoice Generation	Static line items	Dynamic seat count calculations per billing period

The proration calculations become significantly more complex as they must handle both time-based and quantity-based changes. When a customer adds five users mid-cycle, the system calculates the prorated charge based on the remaining days in the billing period multiplied by the per-seat rate for those five additional users.

Custom Contract Support Architecture

Enterprise customers often negotiate custom pricing terms that don't fit standard plan structures. Supporting custom contracts requires a flexible pricing engine that can handle negotiated rates, volume commitments, and custom billing schedules while maintaining the integrity of the standard billing workflows.

Decision: Contract-Based Pricing Architecture

- **Context:** Enterprise customers require custom pricing terms, volume commitments, and specialized billing arrangements that exceed standard plan capabilities
- **Options Considered:**
 1. Extend existing plan system with custom fields
 2. Create parallel contract management system
 3. Build flexible pricing rule engine
- **Decision:** Implement a flexible pricing rule engine with contract overrides
- **Rationale:** Rule engine provides maximum flexibility while maintaining consistency with existing billing workflows, allows gradual migration of complex pricing logic
- **Consequences:** Increased system complexity but enables unlimited pricing model combinations, requires careful testing of rule interactions

Contract Feature	Implementation Approach	Integration Point
Volume Commitments	Minimum usage guarantees with true-up invoicing	Usage aggregation engine
Negotiated Rates	Custom pricing tiers per contract	Proration calculator
Custom Billing Cycles	Flexible billing anchor dates	Renewal processing engine
Revenue Recognition	Deferred revenue allocation	Invoice generation

The contract management system introduces a `Contract` entity that references the base `Plan` but overrides specific pricing components. The billing engine checks for contract overrides at each calculation point, applying custom rates while maintaining the standard billing workflow structure.

Dynamic Pricing and Market-Based Adjustments

Advanced pricing models increasingly incorporate dynamic elements that respond to market conditions, usage patterns, or customer behavior. This might include seasonal pricing adjustments, loyalty discounts based on tenure, or usage-based discounts that reward high-volume customers.

The implementation requires extending the pricing calculation engine to support conditional logic and temporal pricing rules. The system must evaluate pricing conditions at billing time, apply appropriate adjustments, and maintain an audit trail of pricing decisions for compliance and customer transparency.

Dynamic Pricing Type	Trigger Condition	Calculation Method	Persistence Strategy
Seasonal Adjustments	Date-based rules	Percentage modifier on base price	Versioned pricing rules
Loyalty Discounts	Customer tenure calculation	Tiered discount percentage	Customer pricing history
Volume Incentives	Usage threshold analysis	Progressive discount rates	Usage milestone tracking
Market Adjustments	External data feeds	Algorithm-based price optimization	Real-time pricing cache

Multi-Tenant Architecture

As the subscription billing system scales to serve multiple organizations or business units, the architecture must support tenant isolation, data segregation, and resource allocation while maintaining operational efficiency and cost-effectiveness.

Tenant Isolation Strategies

The current single-tenant design stores all customer data in shared tables with global unique identifiers. Multi-tenant architecture requires careful consideration of data isolation levels, from shared databases with tenant identifiers to completely separate infrastructure per tenant.

Decision: Schema-Per-Tenant with Shared Infrastructure

- **Context:** Need to support multiple tenant organizations with strict data isolation requirements while maintaining operational efficiency
- **Options Considered:**
 1. Shared tables with tenant ID columns
 2. Separate database schema per tenant
 3. Completely isolated infrastructure per tenant
- **Decision:** Separate database schema per tenant with shared application infrastructure
- **Rationale:** Provides strong data isolation without infrastructure multiplication costs, enables tenant-specific customizations while sharing application logic
- **Consequences:** Database connection pooling complexity increases, tenant provisioning requires schema creation, backup and migration strategies become more complex

Isolation Level	Data Security	Operational Complexity	Cost Efficiency	Customization Flexibility
Shared Tables	Medium - relies on application logic	Low	High	Low
Schema Per Tenant	High - database-level isolation	Medium	Medium	Medium
Infrastructure Per Tenant	Highest - complete isolation	High	Low	High

The schema-per-tenant approach requires enhancing the `DatabaseManager` to support dynamic schema selection based on request context. Each database operation must include tenant context, ensuring queries execute against the correct schema. The system maintains a tenant registry that maps tenant identifiers to their corresponding database schemas.

Tenant Provisioning Workflow

Creating new tenants involves provisioning database schemas, initializing default configurations, and setting up billing parameters specific to the tenant's requirements. This process must be automated and reliable to support rapid tenant onboarding.

The tenant provisioning workflow includes several critical steps:

- 1. Schema Creation:** Generate a new database schema with all required tables, indexes, and constraints based on the standard billing system structure
- 2. Configuration Initialization:** Create default billing settings, supported currencies, and system parameters appropriate for the tenant's geographic location and business model
- 3. Administrative User Setup:** Provision initial administrative users with appropriate permissions for tenant management
- 4. Integration Configuration:** Set up connections to the tenant's payment gateways, accounting systems, and other external services
- 5. Data Migration:** If applicable, import existing customer and subscription data from legacy systems with proper validation and reconciliation

Provisioning Step	Duration Estimate	Failure Recovery	Validation Requirements
Schema Creation	30 seconds	Drop and recreate	Table count and constraint verification
Configuration Setup	15 seconds	Configuration rollback	Currency and timezone validation
User Provisioning	10 seconds	User deletion	Authentication system integration
Integration Testing	2 minutes	Configuration reset	Payment gateway connectivity
Data Migration	Variable	Partial rollback options	Data integrity checksums

Cross-Tenant Resource Management

Multi-tenant architecture introduces challenges in resource allocation, performance isolation, and cost attribution. The system must prevent one tenant's heavy usage from impacting others while providing fair resource distribution and accurate cost allocation.

The resource management system implements tenant-aware quotas and rate limiting at multiple levels. Database connection pools maintain per-tenant limits to prevent connection exhaustion. API rate limiting applies tenant-specific thresholds based on their subscription tier or negotiated limits. Background processing queues use tenant-aware scheduling to ensure fair processing distribution.

Tenant-Specific Customizations

Different tenant organizations often require customizations in billing logic, invoice formatting, payment processing workflows, or integration requirements. The multi-tenant architecture must support these customizations without creating maintenance complexity or security vulnerabilities.

The customization framework uses a plugin architecture where tenant-specific logic extends standard billing workflows. Custom pricing calculators, invoice formatters, and payment processors implement standard interfaces while providing tenant-specific behavior. The system loads appropriate customizations based on tenant context during request processing.

Customization Type	Implementation Pattern	Isolation Method	Version Management
Pricing Logic	Strategy pattern with tenant plugins	Interface-based isolation	Plugin versioning with compatibility checks
Invoice Templates	Template engine with tenant themes	Resource namespace separation	Template version control
Payment Workflows	Workflow step overrides	Tenant-specific workflow definitions	Workflow schema validation
Integration Endpoints	Tenant-specific adapters	Configuration-driven routing	Adapter version compatibility

Analytics and Revenue Reporting

The billing system generates vast amounts of financial and operational data that provides valuable insights into business performance, customer behavior, and revenue trends. Building comprehensive analytics capabilities transforms raw billing data into actionable business intelligence.

Revenue Recognition and Financial Reporting

Modern subscription businesses must comply with accounting standards like ASC 606 that require careful revenue recognition practices. The system must track deferred revenue for prepaid subscriptions, recognize revenue over service delivery periods, and handle complex scenarios like plan changes and cancellations.

Revenue recognition begins with the invoice generation process but extends far beyond simple cash collection. When a customer pays for an annual subscription, the system must defer the revenue recognition and allocate it monthly over the service delivery period. Plan upgrades and downgrades require pro-rata revenue adjustments that comply with accounting standards.

Revenue Recognition Scenario	Accounting Treatment	System Implementation	Compliance Requirements
Annual Prepayment	Deferred revenue with monthly recognition	Scheduled revenue allocation jobs	ASC 606 performance obligation tracking
Mid-Cycle Upgrades	Immediate recognition of incremental value	Prorated revenue adjustment calculations	Revenue allocation between performance periods
Service Cancellations	Immediate recognition of earned revenue	Revenue recognition acceleration	Refund liability accounting
Usage-Based Charges	Recognition upon service delivery	Real-time revenue recognition for metered usage	Variable consideration estimation

The revenue recognition engine maintains detailed records of performance obligations, contract modifications, and revenue allocation schedules. Monthly revenue recognition jobs process deferred revenue balances and generate accounting entries that integrate with external financial systems.

Customer Lifecycle Analytics

Understanding customer behavior patterns, churn predictors, and growth opportunities requires comprehensive analysis of subscription lifecycle data. The analytics system must track customer journey metrics, cohort analysis, and predictive indicators that inform business strategy.

Customer lifecycle analytics begin with subscription creation and continue through the entire customer relationship. The system tracks key metrics including customer acquisition cost, lifetime value, monthly recurring revenue, churn rates, and expansion revenue from upgrades and additional usage.

Decision: Real-Time Analytics with Batch Aggregation

- **Context:** Business users need both real-time dashboard updates and complex historical analysis requiring different performance and consistency characteristics
- **Options Considered:**
 1. Pure real-time analytics with stream processing
 2. Batch-only analytics with daily updates
 3. Hybrid real-time dashboards with batch analytical processing
- **Decision:** Implement hybrid architecture with real-time metrics and batch analytical processing
- **Rationale:** Real-time dashboards provide operational visibility while batch processing enables complex analytical queries without impacting billing system performance
- **Consequences:** Increased system complexity but optimal performance characteristics for different use cases, requires data consistency management between real-time and batch systems

Analytics Category	Update Frequency	Data Source	Storage System	Query Patterns
Operational Metrics	Real-time	Transaction logs	Time-series database	Simple aggregations
Cohort Analysis	Daily batch	Customer database	Analytical warehouse	Complex joins and pivots
Revenue Forecasting	Weekly batch	Subscription and usage data	Analytical warehouse	Statistical modeling
Churn Prediction	Daily batch	Customer behavior data	Machine learning platform	Predictive modeling

Usage Pattern Analysis and Optimization

Usage-based billing generates detailed consumption data that reveals customer behavior patterns, feature adoption rates, and optimization opportunities. The analytics system must process high-volume usage events and extract meaningful insights about customer engagement and system utilization.

Usage pattern analysis identifies customers approaching plan limits, features with low adoption rates, and optimization opportunities for both customers and the service provider. The system tracks usage trends over time, compares actual usage against purchased allowances, and identifies customers who might benefit from plan changes.

The usage analytics pipeline processes raw usage events through multiple aggregation stages. Real-time aggregation provides current usage dashboards and quota enforcement. Daily batch processing generates

comprehensive usage reports and trend analysis. Weekly analytical jobs perform cohort analysis and usage pattern recognition.

Business Intelligence and Reporting Framework

The comprehensive reporting framework must serve diverse stakeholder needs, from executive revenue dashboards to detailed customer success team reports. The system must balance query performance, data freshness, and analytical flexibility while maintaining data accuracy and security.

The reporting framework uses a layered architecture with operational data stores, analytical warehouses, and specialized reporting databases. Data flows from the transactional billing system through ETL pipelines that clean, transform, and aggregate information for analytical consumption.

Report Category	Target Audience	Update Frequency	Data Requirements	Performance Characteristics
Executive Dashboards	Leadership team	Real-time	High-level KPIs and trends	Fast loading, simple visualizations
Financial Reports	Finance team	Daily	Detailed revenue and cost data	Complex calculations, audit trails
Customer Success	Account managers	Real-time	Individual customer metrics	Drill-down capabilities, alert integration
Product Analytics	Product team	Weekly	Feature usage and adoption	Statistical analysis, cohort comparisons

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
Multi-Tenant Database	PostgreSQL with schemas	Amazon RDS with cross-region replication
Analytics Engine	PostgreSQL analytical queries	Apache Spark with Delta Lake
Real-Time Metrics	Redis with time-series data	InfluxDB with Grafana dashboards
Data Pipeline	Python ETL scripts	Apache Airflow with dbt transformations
Reporting Layer	Django admin with custom views	Tableau or Looker integration
Contract Management	JSON configuration in database	Dedicated contract service with approval workflows

File Structure for Extensions

```
subscription-system/
├── core/
│   ├── billing/                      # Existing billing components
│   ├── usage/                        # Existing usage tracking
│   └── invoicing/                   # Existing invoice generation
├── extensions/
│   ├── advanced_pricing/
│   │   ├── seat_based_billing.py    # Seat-based pricing engine
│   │   ├── contract_manager.py     # Custom contract support
│   │   └── dynamic_pricing.py      # Market-based pricing
│   ├── multi_tenant/
│   │   ├── tenant_manager.py       # Tenant provisioning and management
│   │   ├── schema_manager.py       # Database schema management
│   │   └── resource_manager.py     # Cross-tenant resource allocation
│   └── analytics/
│       ├── revenue_recognition.py # ASC 606 compliant revenue tracking
│       ├── customer_analytics.py  # Lifecycle and churn analysis
│       └── usage_analytics.py     # Usage pattern analysis
└── reporting/
    ├── dashboards/                 # Real-time dashboard components
    ├── etl/                         # Data pipeline jobs
    └── exports/                     # Report generation and export
└── migrations/
    ├── multi_tenant_setup/         # Schema setup for multi-tenancy
    └── analytics_tables/           # Analytical warehouse setup
```

Advanced Pricing Infrastructure

```
"""
Advanced pricing engine supporting seat-based billing and custom contracts.

Extends the existing PricingModel enum and PricingTier structure.

"""

from enum import Enum

from typing import Dict, List, Optional, Decimal

from datetime import datetime, date

from dataclasses import dataclass


class AdvancedPricingModel(Enum):

    SEAT_BASED = "seat_based"

    CONTRACT_CUSTOM = "contract_custom"

    DYNAMIC_PRICING = "dynamic_pricing"

    VOLUME_COMMITMENT = "volume_commitment"


@dataclass

class SeatConfiguration:

    """Configuration for seat-based billing plans"""

    base_seat_count: int

    base_price_cents: int

    additional_seat_price_cents: int

    seat_types: Dict[str, int] # seat_type -> price_per_seat_cents

    proration_behavior: str # "immediate", "next_cycle", "anniversary"

    minimum_seats: Optional[int] = None

    maximum_seats: Optional[int] = None


@dataclass

class ContractTerms:
```

```
"""Custom contract pricing terms that override standard plans"""

contract_id: str

customer_id: str

base_plan_id: str

effective_date: date

expiration_date: Optional[date]

custom_pricing_rules: List[Dict] # Flexible rule definitions

volume_commitments: Dict[str, Decimal] # usage_type -> minimum_quantity

billing_schedule: str # "monthly", "quarterly", "annual", "custom"

revenue_recognition_method: str


class AdvancedPricingEngine:

    """
    Extends the existing pricing engine to support advanced pricing models.

    Integrates with existing Plan and PricingTier structures.

    """

    def __init__(self, base_pricing_engine, contract_manager, seat_tracker):

        # TODO: Initialize with existing pricing engine for fallback

        # TODO: Store references to contract manager and seat tracking

        pass


    def calculate_subscription_charge(self, subscription_id: str,
                                      billing_period_start: datetime,
                                      billing_period_end: datetime) -> Money:

        """
        Calculate total subscription charge including advanced pricing.
        """
```

Steps:

1. Load subscription and determine pricing model type
2. Check for custom contract overrides
3. Apply appropriate pricing calculation method
4. Handle seat-based calculations if applicable
5. Apply any dynamic pricing adjustments
6. Return final calculated amount

"""

```
# TODO: Implement seat-based calculation logic
```

```
# TODO: Check for contract overrides
```

```
# TODO: Apply dynamic pricing rules
```

```
# TODO: Integrate with existing proration logic
```

```
pass
```

```
def calculate_seat_based_charge(self, subscription, seat_config: SeatConfiguration,  
                                current_seats: int, billing_days: int) -> Money:
```

"""

Calculate charges for seat-based billing model.

Steps:

1. Calculate base seat charge (included seats)
2. Calculate additional seat charges
3. Apply seat type specific pricing
4. Handle mid-cycle seat changes with proration
5. Apply minimum/maximum seat constraints

"""

```
# TODO: Implement seat-based pricing calculation

# TODO: Handle different seat types with different rates

# TODO: Calculate proration for mid-cycle seat changes

pass

def apply_contract_pricing(self, base_amount: Money, contract: ContractTerms) -> Money:
    """
    Apply custom contract pricing rules to override standard pricing.

    Steps:
    1. Evaluate contract rule conditions
    2. Apply percentage discounts or fixed amounts
    3. Handle volume commitment calculations
    4. Validate pricing against contract terms
    5. Generate audit trail for contract pricing
    """

    # TODO: Implement contract rule evaluation engine
    # TODO: Handle volume commitment true-up calculations
    # TODO: Apply custom billing schedule adjustments
    pass
```

Multi-Tenant Infrastructure

```
"""
Multi-tenant architecture supporting schema-per-tenant isolation
with shared application infrastructure.

"""

from typing import Dict, Optional, List
import threading
from contextlib import contextmanager
from dataclasses import dataclass

@dataclass
class TenantConfiguration:

    """Configuration settings specific to a tenant"""

    tenant_id: str
    schema_name: str
    database_url: str
    supported_currencies: List[str]
    default_currency: str
    timezone: str
    billing_settings: Dict
    payment_gateway_config: Dict
    feature_flags: Dict[str, bool]

class TenantContext:

    """Thread-local tenant context for request processing"""

    _local = threading.local()

    @classmethod
```

```
def set_current_tenant(cls, tenant_id: str):

    # TODO: Set thread-local tenant context

    # TODO: Load tenant configuration from cache

    # TODO: Validate tenant is active and accessible

    pass


@classmethod

def get_current_tenant(cls) -> Optional[str]:

    # TODO: Return current thread-local tenant ID

    # TODO: Raise exception if no tenant context set

    pass


@classmethod

def get_tenant_config(cls) -> TenantConfiguration:

    # TODO: Return configuration for current tenant

    # TODO: Cache configuration for performance

    pass


class TenantDatabaseManager:

    """"

    Manages database connections and schema routing for multi-tenant architecture.

    Extends the existing DatabaseManager with tenant-aware operations.

    """

    def __init__(self, base_database_manager):

        # TODO: Store reference to base database manager

        # TODO: Initialize tenant schema registry
```

```
# TODO: Setup connection pooling per tenant
pass

@contextmanager
def tenant_transaction(self, isolation_level=None):
    """
    Create database transaction for current tenant schema.

    Steps:
    1. Get current tenant context
    2. Select appropriate database connection
    3. Set schema search path for tenant
    4. Begin transaction with specified isolation
    5. Yield session to caller
    6. Commit or rollback based on exceptions
    7. Reset connection state
    """

    # TODO: Implement tenant-aware transaction management
    # TODO: Handle schema switching and connection pooling
    # TODO: Ensure proper cleanup on success or failure
    pass

def provision_tenant_schema(self, tenant_config: TenantConfiguration) -> bool:
    """
    Create database schema and initial setup for new tenant.
    """
```

Steps:

```
1. Create database schema with tenant name  
2. Run all migration scripts in new schema  
3. Insert default configuration data  
4. Create initial admin user  
5. Validate schema creation success  
6. Register tenant in tenant registry  
"""  
  
# TODO: Implement schema creation and migration  
  
# TODO: Handle rollback on provision failure  
  
# TODO: Validate schema integrity after creation  
  
pass
```

```
def deprovision_tenant_schema(self, tenant_id: str) -> bool:
```

```
"""
```

```
Safely remove tenant schema and all associated data.
```

Steps:

1. Validate tenant can be safely removed
2. Export tenant data for backup if required
3. Drop tenant schema and all objects
4. Remove tenant from registry
5. Clean up connection pool resources

```
"""
```

```
# TODO: Implement safe tenant removal with backup  
  
# TODO: Validate no active connections to tenant schema  
  
# TODO: Handle cleanup of related resources  
  
pass
```

```
class TenantResourceManager:

    """
    Manages resource allocation and limits across tenants.
    """

    def __init__(self):
        # TODO: Initialize resource tracking and limits
        # TODO: Setup tenant quota configuration
        pass

    def check_tenant_quota(self, tenant_id: str, resource_type: str,
                           requested_amount: int) -> bool:
        """
        Check if tenant can consume additional resources.

        Steps:
        1. Load tenant resource quotas and current usage
        2. Check if requested amount exceeds remaining quota
        3. Apply any burst allowances or temporary increases
        4. Log quota check for monitoring and billing
        5. Return availability decision
        """

        # TODO: Implement quota checking with burst handling
        # TODO: Track resource usage per tenant
        # TODO: Handle quota enforcement and alerting
        pass
```

Analytics and Reporting Infrastructure

```
"""
Analytics and revenue reporting system with real-time metrics
and batch analytical processing.

"""

from typing import Dict, List, Optional, Tuple
from datetime import datetime, date, timedelta
from decimal import Decimal
from dataclasses import dataclass

@dataclass
class RevenueRecognitionSchedule:

    """Schedule for recognizing deferred revenue over time"""

    subscription_id: str
    invoice_id: str
    total_amount_cents: int
    recognition_start_date: date
    recognition_end_date: date
    monthly_recognition_cents: int
    recognized_to_date_cents: int
    remaining_deferred_cents: int

class RevenueRecognitionEngine:

    """
    ASC 606 compliant revenue recognition for subscription billing.

    Handles deferred revenue allocation and recognition scheduling.
    """


```

```
def __init__(self, billing_engine, accounting_integration):

    # TODO: Initialize with billing engine and accounting system

    # TODO: Load revenue recognition policies and schedules

    pass


def create_recognition_schedule(self, invoice: Invoice) ->
List[RevenueRecognitionSchedule]:
    """
    Create revenue recognition schedule for invoice line items.

    Steps:
    1. Analyze each line item for revenue recognition requirements
    2. Determine performance obligation periods
    3. Calculate monthly recognition amounts
    4. Handle proration for partial periods
    5. Create recognition schedule entries
    6. Integrate with accounting system journal entries
    """

    # TODO: Implement ASC 606 compliant revenue recognition
    # TODO: Handle different line item types (subscription, usage, etc.)
    # TODO: Create deferred revenue liability accounts

    pass


def process_monthly_recognition(self, recognition_date: date) -> List[Dict]:
    """
    Process monthly revenue recognition for all active schedules.

```

```
Steps:

1. Load all active recognition schedules for the month
2. Calculate recognition amounts for each schedule
3. Generate accounting journal entries
4. Update recognition schedule progress
5. Handle schedule modifications for plan changes
6. Export entries to accounting system

"""

# TODO: Implement monthly revenue recognition batch job

# TODO: Handle plan changes and cancellations

# TODO: Generate accounting system integration data

pass

class CustomerAnalyticsEngine:

"""

Customer lifecycle analytics including cohort analysis and churn prediction.

"""

def __init__(self, subscription_engine, usage_engine):

    # TODO: Initialize with data sources

    # TODO: Setup analytical data pipeline connections

    pass

def calculate_customer_lifetime_metrics(self, customer_id: str) -> Dict:

"""

Calculate comprehensive customer lifetime value and behavior metrics.


```

```
Steps:

1. Load customer subscription and payment history

2. Calculate total revenue and average monthly revenue

3. Determine customer tenure and renewal rates

4. Analyze usage patterns and feature adoption

5. Calculate customer acquisition cost attribution

6. Generate churn risk indicators

"""

# TODO: Implement LTV calculation with usage correlation

# TODO: Calculate customer health scores

# TODO: Generate expansion opportunity indicators

pass
```

```
def generate_cohort_analysis(self, cohort_period: str,
                             analysis_date: date) -> List[Dict]:
```

"""

Generate cohort retention and revenue analysis.

```
Steps:

1. Group customers by acquisition period (monthly cohorts)

2. Track retention rates over subsequent periods

3. Calculate revenue per cohort over time

4. Identify cohort behavior patterns

5. Generate expansion and contraction metrics

6. Export analysis for business intelligence tools

"""

# TODO: Implement cohort analysis with revenue tracking
```



```
# TODO: Implement usage trend analysis with forecasting  
  
# TODO: Generate plan optimization recommendations  
  
# TODO: Identify customers approaching limits for upselling  
  
pass
```

Milestone Checkpoints

Advanced Pricing Validation:

1. **Seat-Based Billing Test:** Create a seat-based plan and add/remove users mid-cycle. Verify prorated charges calculate correctly: `python test_seat_based_billing.py`
2. **Contract Override Test:** Configure a custom contract with negotiated rates. Verify billing engine applies contract pricing instead of standard plan rates
3. **Dynamic Pricing Test:** Setup seasonal pricing rules and verify they apply correctly during the effective periods

Multi-Tenant Validation:

1. **Schema Isolation Test:** Create multiple tenants and verify data isolation. Query from one tenant should never return another tenant's data
2. **Resource Quota Test:** Configure tenant quotas and verify enforcement. Exceeding quotas should block additional resource consumption
3. **Tenant Provisioning Test:** Provision a new tenant and verify complete setup including schema creation, configuration, and user access

Analytics Validation:

1. **Revenue Recognition Test:** Create subscriptions with deferred revenue and verify recognition schedule accuracy over multiple months
2. **Customer Analytics Test:** Generate cohort analysis reports and verify retention/revenue calculations match manual verification
3. **Usage Pattern Test:** Submit usage events and verify analytical aggregations match real-time tracking within acceptable variance

Common Pitfalls and Solutions

⚠️ Pitfall: Seat Change Proration Complexity Seat-based billing introduces complex proration scenarios where quantity and time-based proration interact. Adding 5 seats mid-cycle requires calculating the prorated charge for the remaining billing period, but the calculation must account for different seat types and pricing tiers.

Solution: Implement seat change proration as a specialized case of the general proration engine, treating seat additions as quantity-based plan changes with time-based proration factors.

⚠ Pitfall: Multi-Tenant Data Leakage The most critical risk in multi-tenant architecture is data leakage between tenants. A missing tenant context check or incorrect schema routing could expose one tenant's sensitive financial data to another tenant.

Solution: Implement defense-in-depth with multiple isolation layers: middleware-level tenant validation, database-level schema enforcement, and application-level data access controls. Every database query must include tenant context validation.

⚠ Pitfall: Revenue Recognition Complexity ASC 606 compliance requires detailed tracking of performance obligations and revenue allocation. Plan changes, cancellations, and usage-based billing create complex scenarios where revenue recognition schedules must be modified mid-stream.

Solution: Treat revenue recognition as an event-driven process where subscription changes trigger recognition schedule updates. Maintain detailed audit trails for all recognition adjustments to support compliance requirements.

⚠ Pitfall: Analytics Performance Impact Real-time analytics queries can impact billing system performance, especially complex analytical queries that scan large datasets. Running cohort analysis during peak billing periods could slow invoice generation.

Solution: Implement read replicas for analytical workloads and use batch processing for complex analysis. Provide real-time metrics through pre-computed aggregations and time-series databases optimized for analytical queries.

Glossary

Milestone(s): All milestones - provides comprehensive definitions of billing terminology, technical concepts, and domain-specific vocabulary used throughout the subscription billing system implementation

Mental Model: Building a Shared Language

Think of this glossary as the **shared dictionary** for your billing system team. Just as a successful engineering team develops common terminology for complex concepts, a billing system requires precise vocabulary to prevent costly misunderstandings. When someone says "proration," everyone must understand whether they mean time-based partial charges, quantity-based adjustments, or both. When discussing "dunning," the team needs to know exactly which payment recovery stage is being referenced.

This shared language becomes critical during incident response. When a customer reports a billing discrepancy at 2 AM, the on-call engineer needs to quickly understand whether the issue involves "usage aggregation boundaries," "proration symmetry violations," or "webhook idempotency failures." Clear terminology accelerates both development and operational troubleshooting.

The terminology is organized into logical groupings: core billing concepts that form the foundation, technical implementation terms that describe how the system works, and operational vocabulary used during monitoring and support. Each term includes context about when and why it's used, helping engineers understand not just what terms mean but when to apply them correctly.

Core Billing and Subscription Terminology

These terms form the fundamental vocabulary of subscription billing systems. Understanding these concepts is essential for anyone working with recurring revenue models, whether in product management, engineering, or customer support roles.

Term	Definition	Context and Usage
subscription billing	Recurring payment system with lifecycle management that automatically charges customers at regular intervals for ongoing service access	The foundational concept encompassing all aspects of recurring revenue collection, from plan selection through renewal cycles
billing anchor	Fixed day for recurring subscription charges, typically set when the subscription is created and maintained throughout its lifecycle	Critical for consistent customer expectations - a subscription created on January 15th should always bill on the 15th of each month
proration	Partial charge calculation for mid-cycle changes, ensuring customers pay only for the portion of service they actually receive	Applied during plan upgrades, downgrades, or partial billing periods; must maintain mathematical precision to avoid customer disputes
grandfathering	Protecting existing customers from plan changes by allowing them to maintain their original pricing and terms indefinitely	Essential for customer retention when updating pricing models; requires plan versioning to maintain legacy terms
plan versioning	Creating new plan versions while maintaining old ones, allowing existing subscribers to keep their original terms	Enables product evolution without disrupting existing customer agreements; prevents forced migrations that could cause churn
feature entitlements	Capabilities unlocked by subscription plans, defining what functionality each tier provides to subscribers	Controls access to premium features, API rate limits, storage quotas, and other plan-specific benefits
dunning management	Systematic payment failure recovery process that attempts to collect overdue payments through escalating actions	Automated workflow typically including payment retries, email notifications, service restrictions, and eventual cancellation
grace period	Time window maintaining service access after payment failure, providing customers opportunity to resolve payment issues	Balances business needs (revenue collection) with customer experience (avoiding service disruption during temporary payment issues)
usage-based billing	Charges based on metered consumption, where customers pay for actual usage beyond their plan's included allowances	Common in SaaS for API calls, storage, bandwidth, or processing time; requires robust usage tracking and aggregation systems
credit balance	Accumulated customer credits available for future use, typically from downgrades, refunds, or promotional credits	Must be applied automatically to reduce invoice amounts; requires careful tracking to prevent revenue recognition issues

Term	Definition	Context and Usage
billing period boundaries	Start and end dates for usage calculations and recurring charges, ensuring consistent aggregation windows	Critical for usage-based billing accuracy; typically aligned with subscription anniversary dates to simplify customer understanding

Pricing and Plan Management Terms

These terms relate to the flexible pricing structures and plan configurations that support diverse business models. They're particularly relevant during Milestone 1 implementation and ongoing plan management operations.

Term	Definition	Context and Usage
pricing tiers	Usage brackets with different rates, allowing volume discounts or premium pricing for high consumption	Enables sophisticated pricing strategies like volume discounts; requires careful tier boundary calculations to avoid customer confusion
tiered pricing	Different rates for different usage levels within a single billing period, with each tier charged separately	Each usage tier has its own rate - first 1000 API calls at \$0.10 each, next 5000 at \$0.08 each, etc.
volume pricing	Single rate applied to total usage once a threshold is reached, typically offering better rates for higher volumes	Simpler than tiered - once you hit 1000 API calls, your entire usage is charged at the volume rate
seat-based billing	Pricing model that scales with number of active users or licenses allocated to the customer account	Common in B2B SaaS; requires tracking active user counts and handling seat additions/removals with appropriate proration
usage allowances	Included quantities in subscription plans before overage charges apply	Base plan might include 10,000 API calls per month; additional usage incurs overage charges at specified rates
overage charges	Fees for usage beyond plan allowances, calculated using tiered or volume pricing structures	Applied monthly during invoice generation; customers should receive usage alerts before reaching overage thresholds
usage metrics	Measurable activities that can be billed, such as API calls, storage consumption, or processing time	Must be clearly defined and consistently measured; forms the basis for usage-based billing calculations
quota enforcement	Limiting usage based on plan allowances to prevent unexpected charges or system abuse	Can be soft limits (allow overage with charges) or hard limits (block usage at threshold)
contract-based pricing	Custom pricing terms negotiated for enterprise customers, overriding standard plan rates	Requires special handling in billing logic to apply negotiated discounts or custom rate structures
dynamic pricing	Market-responsive pricing adjustments based on demand, usage patterns, or external conditions	Advanced feature requiring careful implementation to maintain customer trust and billing predictability

Technical Implementation Vocabulary

These terms describe the technical architecture and implementation patterns used to build reliable billing systems. They're essential for engineering teams working on system design, implementation, and maintenance.

Term	Definition	Context and Usage
financial precision	Accurate monetary calculations using integer cents to avoid floating-point rounding errors	Always store money amounts as integers (cents, pence, etc.) and use <code>Money</code> types for all financial operations
currency precision	Using smallest currency unit to avoid rounding errors in international billing scenarios	Essential for multi-currency support; different currencies have different precision requirements (yen has no fractional units)
idempotency	Ensuring operations can be safely retried without causing duplicate charges or state changes	Critical for webhook processing, payment operations, and any operation that might be retried during failures
state machine	Defined states and transitions for subscription lifecycle management with validation rules	Prevents invalid state transitions and ensures consistent subscription behavior across all system components
audit trail	Complete record of financial operations and state changes for compliance and debugging	Required for financial systems; every billing operation must be logged with timestamp, actor, and change details
optimistic concurrency control	Version-based conflict detection for concurrent updates to prevent data corruption	Uses version numbers to detect when two operations attempt to modify the same entity simultaneously
workflow orchestration	Coordinating multi-step business processes across components with compensation and retry logic	Essential for complex operations like subscription creation, plan changes, and billing cycles
event-driven architecture	System design using asynchronous event communication between loosely coupled components	Enables scalability and resilience; billing events trigger downstream processing without tight coupling
transactional outbox	Pattern for reliable event publishing with database transactions to ensure consistency	Guarantees that database changes and event publications succeed or fail together
compensation pattern	Rollback strategy for distributed operations when part of a multi-step process fails	Implements "saga pattern" for distributed transactions; each step must have a corresponding compensation action
circuit breaker	Failure prevention pattern for external service calls that opens when failure thresholds are exceeded	Protects billing system from cascading failures when payment gateways or other external services become unavailable
eventual consistency	Data consistency model allowing temporary inconsistencies that resolve	Acceptable for some billing scenarios where immediate consistency isn't required, such as

Term	Definition	Context and Usage
	over time	usage reporting dashboards

Data and Storage Concepts

These terms relate to data modeling, persistence, and consistency requirements specific to billing systems. They're crucial for understanding data architecture decisions and debugging data-related issues.

Term	Definition	Context and Usage
usage factor	Fraction of billing cycle remaining, used in proration calculations for time-based adjustments	Calculated as (days_remaining / total_days_in_cycle); critical for accurate proration math
proration symmetry	Mathematical property that upgrade followed by downgrade should net to zero charge	Essential invariant for testing proration logic; prevents customer disputes over billing accuracy
billing cycle integration	Coordination between subscription renewals, usage aggregation, and invoice generation	Ensures all charges for a billing period are captured in the correct invoice
real-time approximation	Fast estimates for dashboards and quotas without full aggregation accuracy	Provides immediate feedback for usage-based services while batch processes calculate precise billing amounts
batch aggregation	Scheduled processing for complex analytical computations and precise billing calculations	Runs during off-peak hours to perform expensive usage aggregation and generate accurate invoices
event deduplication	Preventing duplicate charges for same activity when usage events are reported multiple times	Uses idempotency keys or event IDs to identify and skip duplicate usage reports
time zone consistency	UTC timestamp handling across different time zones to ensure consistent billing periods	All internal processing uses UTC; timezone conversion only for customer-facing displays
calendar arithmetic	Date calculations handling month overflow and leap years in billing cycle calculations	February billing cycles, leap years, and month-end subscriptions require special handling
schema-per-tenant	Database isolation strategy with separate schemas per tenant in multi-tenant systems	Provides strong data isolation while sharing underlying database infrastructure
tenant isolation	Preventing data leakage between tenant organizations in multi-tenant billing systems	Critical for SaaS platforms serving multiple organizations; includes data, processing, and access isolation

Payment and Financial Processing Terms

These terms cover payment processing, financial compliance, and revenue recognition aspects of billing systems. They're particularly important for payment gateway integration and financial reporting.

Term	Definition	Context and Usage
payment gateway integration	Connecting to external payment processors like Stripe, PayPal, or Braintree for charge processing	Requires webhook handling, idempotency protection, and proper error handling for payment failures
webhook processing	Handling asynchronous payment status updates from payment gateways reliably	Must include signature verification, idempotency protection, and retry logic for processing failures
idempotency protection	Preventing duplicate operation processing when webhooks or API calls are retried	Uses idempotency keys to identify and skip operations that have already been processed
payment method tokenization	Storing secure references to payment instruments without handling sensitive card data	PCI compliance requirement; payment processors provide tokens that reference stored payment methods
dunning escalation	Progression through increasingly severe actions during payment failure recovery	Typical stages: retry payment → email notification → service restriction → account suspension → cancellation
revenue recognition	ASC 606 compliant allocation of revenue over service delivery periods for financial reporting	Required for public companies; subscription revenue is typically recognized ratably over the service period
deferred revenue	Liability for services not yet delivered on prepaid subscriptions	Annual subscription payments create deferred revenue that's recognized monthly as service is delivered
churn prediction	Analytical model identifying customers at risk of cancellation for retention efforts	Uses billing history, usage patterns, and support interactions to predict cancellation probability
customer lifetime value	Total revenue expected from customer relationship over its entire duration	Key metric for subscription businesses; influences customer acquisition spending and retention investments
cohort analysis	Grouping customers by acquisition period for retention analysis and business planning	Tracks how customer groups behave over time; essential for understanding business health and growth trends

Error Handling and Debugging Terminology

These terms are essential for troubleshooting billing issues, handling edge cases, and maintaining system reliability in production environments.

Term	Definition	Context and Usage
billing calculation debugging	Systematic approach to identifying and resolving mathematical errors in subscription billing	Involves tracing calculation inputs, intermediate steps, and final outputs to identify discrepancies
state consistency	Alignment between subscription states and related entity states like invoices and payments	Prevents scenarios where subscription shows active but last payment failed; requires cross-entity validation
webhook reliability	Ensuring payment gateway events are consistently processed despite network issues or system failures	Requires idempotency, retry logic, and dead letter queues for failed webhook processing
mathematical invariants	Properties that must hold true across all test scenarios, such as proration symmetry	Used in property-based testing to verify billing logic correctness across many input combinations
currency precision testing	Validation of monetary calculations using integer cents to prevent rounding errors	Ensures all money operations maintain precision and avoid accumulation of rounding errors
billing logic unit tests	Mathematical validation of calculation accuracy for proration, usage charges, and pricing tiers	Tests individual calculation functions with known inputs and expected outputs
integration testing	Validation of component interaction and data flow across system boundaries	Ensures components work correctly together and data flows properly through complete workflows
property-based testing	Automated test generation verifying mathematical invariants across many random inputs	Particularly valuable for billing systems where mathematical correctness is critical
milestone validation checkpoints	Verification criteria for development phase completion with specific acceptance tests	Defines what must work correctly before proceeding to next development milestone
webhook processing reliability	Ensuring payment gateway webhooks are processed exactly once despite retries or failures	Critical for payment status updates; failures can lead to service disruptions or revenue loss

Advanced Architecture and Scalability Terms

These terms relate to sophisticated architectural patterns and scalability considerations for large-scale billing systems serving many customers and high transaction volumes.

Term	Definition	Context and Usage
multi-tenant architecture	System design supporting multiple isolated tenant organizations with shared infrastructure	Enables SaaS platforms to serve many customer organizations while maintaining data isolation and security
resource allocation	Fair distribution of system resources across tenants to prevent noisy neighbor problems	Includes CPU, memory, database connections, and API rate limits; ensures tenant isolation
performance obligation	Distinct service commitment requiring revenue recognition under ASC 606 accounting standards	Each separately identifiable service component may require separate revenue recognition treatment
usage pattern analysis	Examination of consumption trends for optimization opportunities and capacity planning	Identifies customers approaching plan limits, unusual usage spikes, or opportunities for plan upgrades
real-time analytics	Immediate data processing for operational dashboards and customer-facing usage displays	Provides current usage information while batch processes handle precise billing calculations
distributed transaction management	Coordinating consistency across multiple services and databases in billing operations	Uses saga patterns and compensation logic to maintain consistency without distributed locks
eventual consistency patterns	Designing for temporary inconsistencies that resolve through background synchronization	Acceptable for some billing scenarios where immediate consistency isn't required
horizontal scaling strategies	Techniques for scaling billing systems across multiple servers and database shards	Includes sharding by customer ID, tenant isolation, and distributed processing patterns
capacity planning	Forecasting resource requirements based on customer growth and usage patterns	Critical for handling billing cycles, usage processing spikes, and payment processing loads
disaster recovery	Business continuity planning for billing system failures and data center outages	Includes backup strategies, failover procedures, and recovery time objectives for financial systems

Compliance and Regulatory Terms

These terms cover legal, regulatory, and compliance requirements that billing systems must address, particularly in regulated industries or international markets.

Term	Definition	Context and Usage
PCI compliance	Payment Card Industry security standards for handling credit card information	Required when processing payments; affects system architecture and data handling procedures
data residency	Legal requirements for where customer data must be stored and processed geographically	Important for international billing systems serving customers in regulated jurisdictions
audit compliance	Meeting requirements for financial record keeping and audit trail maintenance	Includes immutable audit logs, change tracking, and retention policies for financial transactions
tax calculation	Computing appropriate taxes based on customer location and applicable tax rates	Complex in international contexts; often requires integration with tax calculation services
regulatory reporting	Meeting requirements for financial reporting to regulatory authorities	Includes revenue recognition, tax reporting, and anti-money laundering compliance
customer data protection	Implementing GDPR, CCPA, and other privacy regulations in billing systems	Affects data collection, storage, processing, and customer rights to access and deletion
financial controls	Internal processes ensuring accuracy and preventing fraud in billing operations	Includes segregation of duties, approval workflows, and reconciliation procedures
retention policies	Legal requirements for how long financial records must be maintained	Varies by jurisdiction and industry; affects database design and archival strategies
cross-border payments	Handling international transactions with currency conversion and regulatory compliance	Includes foreign exchange rates, international banking regulations, and tax implications
subscription cancellation rights	Legal requirements for customer cancellation processes and refund policies	Varies by jurisdiction; affects subscription lifecycle design and refund processing

Testing and Quality Assurance Vocabulary

These terms describe testing strategies, quality assurance practices, and validation approaches specific to billing systems where accuracy and reliability are paramount.

Term	Definition	Context and Usage
end-to-end testing	Complete workflow validation from customer signup through payment processing and service delivery	Ensures entire billing process works correctly including external integrations
load testing	Validating system performance under expected and peak transaction volumes	Critical for billing cycles when many customers are processed simultaneously
chaos engineering	Deliberately introducing failures to test system resilience and recovery procedures	Important for billing systems where failures can impact revenue and customer trust
regression testing	Ensuring changes don't break existing billing calculations or customer agreements	Critical when updating pricing logic or adding new features to billing system
A/B testing	Comparing different pricing strategies or billing flows to optimize conversion and retention	Requires careful implementation to avoid billing inconsistencies or customer confusion
canary deployment	Gradual rollout of billing system changes to detect issues before full deployment	Essential for billing systems where bugs can cause widespread customer impact
synthetic monitoring	Automated testing of critical billing workflows in production to detect issues quickly	Includes test transactions, payment processing, and webhook delivery validation
data quality monitoring	Continuous validation of billing data accuracy and consistency	Detects issues like duplicate charges, missing usage events, or calculation errors
reconciliation testing	Comparing system calculations with external sources to ensure accuracy	Includes payment gateway reconciliation, usage source validation, and tax calculation verification
boundary testing	Validating system behavior at edge cases like month boundaries, leap years, and timezone changes	Critical for billing systems where edge cases can cause calculation errors or service disruptions

Implementation Guidance

This implementation section provides practical resources for maintaining consistent terminology and building shared understanding across your billing system team.

Technology Recommendations

Component	Simple Option	Advanced Option
Terminology Management	Shared Markdown glossary in repository	Confluence or Notion with search and cross-references
Code Documentation	Inline comments with term definitions	Generated API documentation with glossary links
Team Communication	Slack channel for terminology questions	Dedicated wiki with examples and usage guidelines
Customer Communication	Simple glossary page on website	Interactive help system with contextual definitions

Recommended File Structure

```
project-root/
  docs/
    glossary.md           ← comprehensive term definitions
    terminology-guide.md   ← usage guidelines for team
    customer-glossary.md   ← customer-facing definitions
  src/
    shared/
      types/
        billing_types.py     ← standardized type definitions
        money.py              ← Money type with precise arithmetic
      constants/
        billing_constants.py  ← standard enum values and constants
      exceptions/
        billing_exceptions.py ← standardized error types
```

Terminology Validation Tools

Automated Terminology Checker:

```
class TerminologyValidator:
```

PYTHON

```
    """Validates consistent term usage across codebase and documentation."""
```

```
    def __init__(self, glossary_path: str):
```

```
        # TODO: Load glossary terms and approved aliases
```

```
        # TODO: Build term validation rules and patterns
```

```
        # TODO: Initialize spell checker with billing vocabulary
```

```
    pass
```

```
    def validate_code_comments(self, file_path: str) -> List[ValidationResult]:
```

```
        """Check code comments for consistent terminology usage."""
```

```
        # TODO: Parse code comments and docstrings
```

```
        # TODO: Check against glossary terms and flag inconsistencies
```

```
        # TODO: Suggest correct terms for common misspellings
```

```
        # TODO: Return validation results with line numbers and suggestions
```

```
    pass
```

```
    def validate_documentation(self, doc_path: str) -> List[ValidationResult]:
```

```
        """Check documentation for terminology consistency."""
```

```
        # TODO: Parse markdown/text content
```

```
        # TODO: Identify billing terminology usage
```

```
        # TODO: Flag undefined terms or inconsistent usage
```

```
        # TODO: Suggest approved alternatives from glossary
```

```
    pass
```

Team Communication Templates:

```
# Standard comment template for complex billing logic
```

PYTHON

```
"""
```

Calculates proration for subscription plan changes.

Billing Terms:

- proration: partial charge calculation for mid-cycle changes
- usage factor: fraction of billing cycle remaining
- billing anchor: fixed day for recurring subscription charges

Args:

```
old_plan: Current subscription plan with pricing
```

```
new_plan: Target plan for upgrade/downgrade
```

```
change_date: When plan change becomes effective
```

Returns:

```
ProrationResult with charge_amount, credit_amount, net_amount
```

Invariants:

- Maintains proration symmetry (upgrade then downgrade nets zero)
- Preserves billing anchor date throughout plan changes

```
"""
```

Common Terminology Mistakes

⚠ Pitfall: Inconsistent Money Terminology Teams often mix "amount," "price," and "cost" inconsistently.

Establish clear conventions: use "amount" for customer-owed money, "price" for plan rates, and "cost" for business expenses. This prevents confusion in code reviews and customer communication.

⚠ Pitfall: Overloaded State Terms Terms like "active," "enabled," and "running" get used interchangeably for subscription states. Use the exact `SubscriptionStatus` enum values (`active`, `past_due`, `cancelled`) consistently across all contexts to prevent state machine bugs.

⚠ Pitfall: Vague Usage Terminology Distinguishing between "usage events," "usage metrics," and "usage aggregations" is critical for usage-based billing. Usage events are raw activity records, metrics are measurable quantities, and aggregations are billing-period summaries.

Milestone Checkpoints

Terminology Consistency Validation:

- Run terminology checker across codebase: `python scripts/check_terminology.py`
- Expected: No inconsistent term usage, all billing terms match glossary
- Manual verification: Code reviews should reference specific glossary terms
- Signs of issues: Mixed terminology in related functions, undefined terms in comments

Customer-Facing Language Validation:

- Review API error messages and documentation for consistent terminology
- Expected: All customer communications use approved terms from customer glossary
- Manual verification: Support team can explain any billing term using glossary definitions
- Signs of issues: Customer questions about unclear billing terminology

This comprehensive glossary serves as the authoritative reference for all billing system terminology. Teams should reference it during code reviews, architectural discussions, and customer communication to maintain consistency and clarity throughout the system lifecycle.