

Payment Gateway Integration: Design Document

Overview

This system builds a secure payment processing gateway that handles credit card transactions, maintains PCI compliance, and ensures reliable payment state management through idempotency controls and webhook reconciliation. The key architectural challenge is maintaining data consistency across multiple external payment providers while handling the complexities of asynchronous payment flows, dispute management, and regulatory compliance.

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 - understanding the payment processing domain is essential before implementing payment intents (Milestone 1), payment processing (Milestone 2), refunds/disputes (Milestone 3), and webhook reconciliation (Milestone 4).

Payment processing represents one of the most complex and regulated domains in software engineering, where the stakes of implementation errors extend far beyond typical application bugs. A failed payment can mean lost revenue, regulatory violations, or compromised customer trust. Unlike many software systems where eventual consistency and best-effort delivery are acceptable, payment systems demand strict consistency, cryptographic security, and regulatory compliance that can make or break a business.

The fundamental challenge lies in orchestrating a distributed transaction across multiple parties - the customer's bank, the merchant's bank, payment networks like Visa or Mastercard, and various intermediary processors - while maintaining data integrity and handling the inevitable network partitions, timeouts, and edge cases that occur in real-world financial infrastructure. Each party in this ecosystem has different data models, error handling approaches, and timing requirements, yet the end customer expects a seamless experience where their payment either succeeds completely or fails cleanly with clear error messaging.

Modern payment processing must also navigate an increasingly complex regulatory landscape including PCI DSS compliance for handling card data, Strong Customer Authentication (SCA) requirements under PSD2 in Europe, and various anti-money laundering (AML) regulations. These requirements aren't optional features that can be added later - they must be designed into the system architecture from the beginning, influencing everything from data storage patterns to API design decisions.

Payment Processing Mental Model

Think of payment processing like orchestrating a complex banking transaction that involves multiple banks, but instead of happening face-to-face with a teller who can handle unexpected situations, everything must be automated through a series of computer-to-computer messages sent over unreliable networks. In the physical world, if you write a check to pay for groceries, the cashier can immediately see if your signature matches, but they won't know if you have sufficient funds until later when the check clears through the banking system. Similarly, when a customer enters their credit card information on

a website, the initial response only tells you whether the card details are valid and the issuing bank initially approves the transaction - the actual movement of money happens later through a settlement process.

The **payment intent** concept maps to the moment when a customer decides to buy something and provides their payment information, but before any money actually moves. Think of this like signing a contract that says "I intend to pay \$50 for this item" - it's a commitment, but not yet a completed transaction. The merchant can use this intent to reserve the item for the customer and begin preparing for shipment, but they haven't actually been paid yet. This separation between intent and execution is crucial because many things can still go wrong: the customer's bank might decline the transaction upon further review, the customer might need to complete additional authentication steps, or the merchant might discover the item is out of stock.

3D Secure authentication resembles the process of a bank calling you to verify a large or unusual transaction. Just as your bank might call and ask "Did you really try to spend \$2000 at a electronics store in another state?", 3D Secure redirects the customer to their bank's website where they must prove their identity through a password, SMS code, or biometric verification. The merchant never sees this authentication process - they just know that the customer either passed or failed the verification, similar to how a store clerk doesn't hear your conversation with the bank when they call to verify a large purchase.

The **webhook reconciliation** process functions like the end-of-day settlement that happens in traditional banking. Throughout the day, banks keep track of all the promises to pay (authorizations) and actual fund transfers (captures), but they don't reconcile their books until later. Similarly, payment processors send webhook notifications to update merchants about status changes that happened after the initial API response. A payment might initially appear successful, but a webhook hours later might indicate it was actually declined due to fraud detection, or a pending payment might transition to completed once it fully settles through the banking networks.

Idempotency in payments mirrors the banking principle that you cannot accidentally deposit the same check twice. In the physical world, banks have check numbers and can detect duplicate deposits. In payment processing, idempotency keys serve the same purpose - if a customer accidentally clicks "Pay Now" twice due to a slow internet connection, the second request should be safely ignored rather than charging their card twice. This isn't just about preventing duplicate charges; it's about maintaining the fundamental accounting principle that every financial transaction should happen exactly once.

The **dispute and chargeback** process resembles the consumer protection mechanisms in traditional banking. When a customer calls their bank to report an unauthorized charge or defective merchandise, the bank can forcibly reverse the transaction and require the merchant to prove the charge was legitimate. This happens outside of the normal payment flow - the merchant only learns about it when they receive a chargeback notification, similar to how a check might bounce several days after you deposited it.

Existing Payment Gateway Approaches

Payment gateway implementation involves several architectural patterns, each representing different trade-offs between security, user experience, development complexity, and regulatory compliance. Understanding these patterns is essential before choosing an implementation approach, as the decision affects everything from PCI compliance requirements to the customer checkout experience.

Decision: Payment Gateway Integration Pattern

- **Context:** Applications need to accept payments while minimizing security risks and development complexity. Different integration patterns offer varying levels of control over the payment experience versus security and compliance burden.
- **Options Considered:** Hosted payment pages (redirect model), direct API integration (server-to-server), embedded payment forms (client-side tokenization)
- **Decision:** Hybrid approach using embedded forms with server-side completion for optimal balance of security and user experience
- **Rationale:** Provides seamless user experience without handling raw card data, reduces PCI scope while maintaining control over checkout flow, enables advanced features like saved payment methods
- **Consequences:** Requires careful implementation of client-side tokenization and server-side payment completion, but offers maximum flexibility for future enhancements

Integration Pattern	Security Model	PCI Compliance Scope	User Experience	Development Complexity	Control Level
Hosted Payment Pages	Provider handles all card data	Minimal (SAQ A)	Redirect disruption	Low	Limited customization
Direct API Integration	Server handles raw card data	Full compliance required (SAQ D)	Seamless	High	Complete control
Embedded Forms	Client-side tokenization	Moderate (SAQ A-EP)	Seamless	Medium	High customization

Hosted Payment Pages represent the most secure but least flexible approach. In this pattern, the merchant's website redirects customers to the payment processor's hosted checkout page, similar to how PayPal checkout works. The customer completes their payment on the processor's domain, then returns to the merchant site with a success or failure result. This approach minimizes the merchant's PCI compliance burden since they never handle card data directly, requiring only the simplest Self-Assessment Questionnaire (SAQ A). However, the redirect disrupts the user experience and provides limited customization options. Many customers abandon purchases during redirects, particularly on mobile devices where the transition feels jarring.

The security benefit stems from the fact that sensitive card data never touches the merchant's servers or even their web pages. The payment processor bears full responsibility for securing card data, handling PCI compliance, and managing fraud detection. For small businesses or startups with limited security expertise, this approach offers the fastest time to market with minimal ongoing compliance overhead. However, the merchant sacrifices control over the checkout experience and cannot easily implement features like saved payment methods or subscription billing without additional integration work.

Direct API Integration provides maximum control and flexibility by allowing the merchant's server to communicate directly with the payment processor's APIs using raw card data. The customer enters their payment information on the merchant's website, which submits it directly to the merchant's server for processing. This approach enables seamless user experiences, complete customization of the payment flow, and easy implementation of advanced features like subscription billing or marketplace payments.

However, direct integration comes with significant security and compliance responsibilities. The merchant must achieve full PCI DSS compliance (SAQ D), which requires extensive security controls including network segmentation, vulnerability scanning, penetration testing, and formal security policies. Any server that processes, stores, or transmits card data must be secured according to PCI standards, which can cost tens of thousands of dollars annually for compliance assessments and security infrastructure.

The technical complexity extends beyond just API integration to include secure data handling, encryption key management, and comprehensive logging for compliance audits. Many organizations underestimate the ongoing operational burden of maintaining PCI compliance, particularly as their infrastructure evolves and scales. A single security vulnerability or compliance gap can result in fines, increased processing fees, or loss of payment processing privileges.

Embedded Payment Forms offer a hybrid approach that balances security with user experience through client-side tokenization. The customer enters their payment information into form fields that appear to be part of the merchant's website but are actually served from the payment processor's domain through iframe embedding or JavaScript widgets. When the customer submits the form, their card data goes directly to the payment processor, which returns a token representing the payment method. The merchant's server then uses this token to complete the payment without ever handling raw card data.

This approach reduces PCI scope to SAQ A-EP, which is significantly simpler than full compliance but more involved than SAQ A. The merchant must secure their web pages that host the embedded forms and ensure they don't intercept or store card data. The user experience remains seamless since customers never leave the merchant's website, while the security model ensures sensitive data never reaches the merchant's infrastructure.

The implementation complexity lies in properly integrating the client-side tokenization with server-side payment completion. The merchant must handle the asynchronous nature of client-side tokenization, validate that tokens correspond to actual payment intents, and implement proper error handling when tokenization fails. Additionally, advanced features like saved payment methods require careful coordination between client-side token management and server-side customer records.

Design Insight: The Token Security Model Tokenization represents a fundamental shift in payment security thinking. Instead of treating card data as something to be protected through encryption and access controls, tokenization eliminates the sensitive data entirely from the merchant's environment. The token is cryptographically useless outside the specific merchant-processor relationship, so even if it's compromised, it cannot be used to make unauthorized payments elsewhere.

Security Consideration	Hosted Pages	Direct Integration	Embedded Forms
Card data handling	Never touches merchant	Raw data on servers	Client-side tokenization
PCI compliance burden	Minimal annual questionnaire	Full compliance program	Moderate questionnaire
Data breach risk	Limited to transaction IDs	Complete card data exposure	Token exposure only
Fraud liability	Shared with processor	Primary merchant responsibility	Shared with processor
Implementation timeline	Days to weeks	Months to years	Weeks to months

Cross-cutting Security Considerations affect all integration patterns and must be carefully evaluated regardless of the chosen approach. Transport layer security requires TLS 1.2 or higher for all payment-related communications, with proper certificate validation and HSTS implementation. Session management must prevent session fixation attacks and ensure

payment tokens are properly scoped to authenticated users. Client-side security involves Content Security Policy (CSP) headers to prevent script injection and proper CORS configuration for cross-domain API calls.

Regulatory Compliance extends beyond PCI DSS to include regional requirements like Strong Customer Authentication (SCA) under PSD2 in Europe, which mandates two-factor authentication for most online payments. The chosen integration pattern affects how easily these requirements can be implemented. Hosted pages typically include built-in compliance features, while direct integration requires custom implementation of authentication flows and regulatory reporting.

Scalability and Performance characteristics vary significantly between patterns. Hosted pages eliminate payment-related load from the merchant's infrastructure but introduce dependency on the processor's availability and performance. Direct integration puts full load on the merchant's servers but provides complete control over performance optimization. Embedded forms balance these concerns by handling the most resource-intensive operations (card validation, fraud checking) on the processor's infrastructure while keeping the merchant in control of the overall checkout flow.

The choice of integration pattern fundamentally shapes the rest of the payment system architecture. A system designed for hosted pages cannot easily migrate to direct integration due to the different security and compliance requirements. Similarly, a direct integration approach requires substantially different infrastructure and operational procedures compared to embedded forms. This decision should be made early in the design process with careful consideration of long-term business requirements, technical capabilities, and risk tolerance.

Implementation Guidance

This section establishes the foundational understanding necessary for implementing a secure payment gateway. The following guidance provides concrete technology recommendations and structural patterns that will support all subsequent payment processing components.

A. Technology Stack Recommendations

Component	Simple Option	Advanced Option	Rationale
Web Framework	Flask with SQLAlchemy	FastAPI with Pydantic	Start simple, migrate to FastAPI for better async support
Database	PostgreSQL with psycopg2	PostgreSQL with asyncpg	PostgreSQL provides ACID transactions essential for payments
HTTP Client	requests library	httpx with async support	Synchronous initially, async for high-volume processing
Validation	Manual validation	Pydantic data models	Strong typing prevents payment amount errors
Logging	Python logging module	Structured logging with loguru	Compliance requires detailed audit trails
Task Queue	None (synchronous)	Celery with Redis	Background processing for webhook handling

B. Project Structure Foundation

```

payment_gateway/
├── app/
│   ├── __init__.py
│   ├── main.py          # FastAPI application entry point
│   ├── config.py        # Configuration management
│   ├── database.py      # Database connection and models
│   └── exceptions.py   # Payment-specific exception classes
├── app/models/
│   ├── __init__.py
│   ├── payment_intent.py    # PaymentIntent model (Milestone 1)
│   ├── charge.py           # Charge and payment processing (Milestone 2)
│   ├── refund.py            # Refund and dispute models (Milestone 3)
│   └── webhook_event.py    # Webhook event tracking (Milestone 4)
├── app/services/
│   ├── __init__.py
│   ├── payment_intent_service.py  # Business logic for payment intents
│   ├── payment_processor.py      # External provider integration
│   ├── refund_service.py         # Refund processing logic
│   └── webhook_service.py       # Webhook processing and verification
├── app/api/
│   ├── __init__.py
│   ├── payment_intents.py      # REST endpoints for payment intents
│   ├── charges.py              # Payment processing endpoints
│   ├── refunds.py               # Refund management endpoints
│   └── webhooks.py             # Webhook receiver endpoint
├── app/utils/
│   ├── __init__.py
│   ├── idempotency.py          # Idempotency key handling
│   ├── security.py             # Signature verification utilities
│   └── validators.py           # Input validation helpers
└── tests/
    ├── conftest.py            # Pytest configuration and fixtures
    ├── test_payment_intents.py # Milestone 1 tests
    ├── test_charges.py         # Milestone 2 tests
    ├── test_refunds.py         # Milestone 3 tests
    └── test_webhooks.py        # Milestone 4 tests
└── migrations/               # Database schema migrations
└── docs/                     # API documentation
└── requirements.txt          # Python dependencies

```

C. Infrastructure Starter Code

Configuration Management (`app/config.py`):

```
import os

from typing import Optional

from pydantic import BaseSettings, validator


class Settings(BaseSettings):
    """Application configuration with validation and type safety."""

    # Database configuration

    database_url: str = "postgresql://localhost:5432/payment_gateway"

    database_pool_size: int = 20

    # Payment provider configuration

    stripe_secret_key: str

    stripe_webhook_secret: str

    stripe_api_version: str = "2023-10-16"

    # Security configuration

    secret_key: str

    algorithm: str = "HS256"

    # Application settings

    environment: str = "development"

    log_level: str = "INFO"

    @validator('stripe_secret_key')

    def validate_stripe_key(cls, v):

        if not v.startswith(('sk_test_', 'sk_live_')):

            raise ValueError('Invalid Stripe secret key format')

        return v
```

```
@validator('secret_key')

def validate_secret_key(cls, v):
    if len(v) < 32:
        raise ValueError('Secret key must be at least 32 characters')
    return v

class Config:

    env_file = ".env"

    settings = Settings()
```

Database Connection (app/database.py):

PYTHON

```
from sqlalchemy import create_engine, MetaData

from sqlalchemy.ext.declarative import declarative_base

from sqlalchemy.orm import sessionmaker

from sqlalchemy.pool import QueuePool

from app.config import settings

# Create database engine with connection pooling

engine = create_engine(

    settings.database_url,

    poolclass=QueuePool,

    pool_size=settings.database_pool_size,

    max_overflow=30,

    pool_pre_ping=True, # Validates connections before use

    echo=settings.environment == "development"

)

# Session factory for database operations

SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

# Base class for SQLAlchemy models

Base = declarative_base()

# Metadata for migrations and schema management

metadata = MetaData()

def get_db():

    """Dependency injection for database sessions."""

    db = SessionLocal()

    try:

        yield db

    finally:

        db.close()
```

Payment-Specific Exceptions (`app/exceptions.py`):

```
from typing import Optional, Dict, Any

class PaymentGatewayException(Exception):

    """Base exception for all payment gateway errors."""

    def __init__(self, message: str, error_code: Optional[str] = None,
                 details: Optional[Dict[str, Any]] = None):
        self.message = message
        self.error_code = error_code
        self.details = details or {}
        super().__init__(self.message)

class IdempotencyError(PaymentGatewayException):

    """Raised when idempotency key conflicts occur."""

    pass

class PaymentProcessingError(PaymentGatewayException):

    """Raised when external payment processor returns an error."""

    def __init__(self, message: str, provider_error_code: Optional[str] = None,
                 provider_message: Optional[str] = None):
        self.provider_error_code = provider_error_code
        self.provider_message = provider_message
        super().__init__(message, provider_error_code, {
            'provider_message': provider_message
        })

class ValidationError(PaymentGatewayException):

    """Raised when input validation fails."""

    pass

class WebhookVerificationError(PaymentGatewayException):
```

```
"""Raised when webhook signature verification fails."""  
pass
```

D. Core Validation Utilities (`app/utils/validators.py`):

```
import re

from decimal import Decimal, InvalidOperation

from typing import Union, Optional


def validate_amount(amount: Union[int, str, Decimal]) -> int:

    """
    Validate and convert payment amount to cents (smallest currency unit).
    """
```

Args:

amount: Amount in dollars (string/Decimal) or cents (int)

Returns:

Amount in cents as integer

Raises:

ValidationError: If amount is invalid

"""

```
# TODO 1: Handle integer input (already in cents)

# TODO 2: Handle string/Decimal input (convert from dollars to cents)

# TODO 3: Validate amount is positive and not zero

# TODO 4: Validate amount doesn't exceed maximum (e.g., $1,000,000)

# TODO 5: Round to nearest cent to handle floating point precision

pass
```

```
def validate_currency(currency: str) -> str:
```

"""

Validate ISO 4217 currency code.

Args:

currency: Three-letter currency code (e.g., 'USD', 'EUR')

```
Returns:
```

```
    Uppercase currency code
```

```
Raises:
```

```
    ValidationError: If currency code is invalid
```

```
"""
```

```
# TODO 1: Convert to uppercase
```

```
# TODO 2: Validate exactly 3 alphabetic characters
```

```
# TODO 3: Check against supported currency list
```

```
pass
```

```
def validate_idempotency_key(key: str) -> str:
```

```
"""
```

```
Validate idempotency key format and length.
```

```
Args:
```

```
    key: Client-provided idempotency key
```

```
Returns:
```

```
    Validated idempotency key
```

```
Raises:
```

```
    ValidationError: If key format is invalid
```

```
"""
```

```
# TODO 1: Check key is not empty
```

```
# TODO 2: Validate length (recommended: 22-255 characters)
```

```
# TODO 3: Ensure key contains only safe characters (alphanumeric, hyphens, underscores)
```

```
# TODO 4: Prevent SQL injection patterns
```

```
pass
```

E. Language-Specific Best Practices

For Python payment processing implementations:

- **Decimal Arithmetic:** Always use `decimal.Decimal` for monetary amounts, never `float`. Floating-point arithmetic introduces rounding errors that can cause accounting discrepancies.
- **Environment Variables:** Store all secrets in environment variables, never in code. Use `python-dotenv` for local development and proper secret management in production.
- **Database Transactions:** Wrap all payment operations in database transactions using SQLAlchemy's session management. Payment operations must be atomic.
- **Input Validation:** Use Pydantic models for all API input validation. Payment systems cannot tolerate invalid data reaching business logic.
- **Error Logging:** Log all payment errors with sufficient detail for debugging, but never log sensitive data like full card numbers or CVV codes.
- **Time Handling:** Use UTC for all timestamps and convert to local time only for display. Payment reconciliation across time zones requires consistent time handling.

F. Security Implementation Checklist

Security Control	Implementation	Validation Method
TLS Configuration	Enforce TLS 1.2+ for all endpoints	SSL Labs scan
Input Validation	Pydantic models for all inputs	Fuzzing tests
SQL Injection Prevention	SQLAlchemy ORM with parameterized queries	Static analysis
Secret Management	Environment variables with validation	Security audit
CORS Configuration	Strict origin policy for API endpoints	Browser testing
Rate Limiting	Per-IP and per-user rate limits	Load testing

G. Development Environment Setup

1. **Database Setup:** Install PostgreSQL and create development database with proper encoding (UTF-8) and collation for international character support.
2. **Environment Configuration:** Create `.env` file with all required settings. Use strong, randomly generated values for secrets in development.
3. **Dependency Management:** Use `pip-tools` or `poetry` for reproducible dependency management. Payment systems require stable, auditable dependencies.
4. **Testing Infrastructure:** Configure pytest with database fixtures that create/tear down test data for each test. Payment tests must be isolated and repeatable.
5. **Linting and Formatting:** Configure `black`, `isort`, and `flake8` for consistent code style. Financial code requires high readability standards.

H. Milestone Validation Checkpoints

After completing each milestone, validate progress with these concrete checks:

Milestone 1 Checkpoint (Payment Intents):

- Create payment intent via API with valid idempotency key
- Attempt duplicate creation with same key - should return original intent
- Verify payment intent state transitions are properly validated
- Check database constraints prevent invalid state changes

Milestone 2 Checkpoint (Payment Processing):

- Process test payment with valid card details
- Trigger 3D Secure flow with test card requiring authentication
- Verify payment method tokenization prevents card data storage
- Test payment confirmation after successful authentication

Milestone 3 Checkpoint (Refunds/Disputes):

- Process full refund against completed payment
- Process partial refund with proper amount validation
- Simulate chargeback notification and verify dispute creation
- Test refund state transitions and accounting accuracy

Milestone 4 Checkpoint (Webhooks):

- Receive webhook with valid signature - should process successfully
- Send webhook with invalid signature - should reject
- Test webhook idempotency with duplicate event IDs
- Verify state reconciliation detects and corrects discrepancies

Each checkpoint should include both automated tests and manual verification steps to ensure the implementation meets production readiness standards.

Goals and Non-Goals

Milestone(s): Foundation for all milestones - clearly scoped requirements guide implementation decisions across payment intents (Milestone 1), payment processing (Milestone 2), refunds and disputes (Milestone 3), and webhook reconciliation (Milestone 4).

Setting clear boundaries for a payment gateway implementation is like drawing a fence around a construction site. Without explicit boundaries, scope creep inevitably leads to an over-engineered system that tries to solve every conceivable payment problem. Payment processing touches on numerous complex domains—fraud detection, regulatory compliance, international banking, subscription management, marketplace settlements, and more. Each of these areas could consume months of development effort on its own. This section establishes what our payment gateway will accomplish within a reasonable implementation timeline, what quality attributes it must satisfy, and crucially, what advanced features we explicitly defer to future iterations.

The mental model for goal-setting in payment systems is similar to designing a bridge. Functional goals define what the bridge carries (cars, pedestrians, weight limits). Non-functional goals specify how well it carries them (earthquake resistance, expected lifespan, maintenance requirements). Non-goals clarify what the bridge won't support (freight trains,

aircraft landings, unlimited weight). Without this clarity, engineers either under-build (bridge collapses) or over-build (bridge costs 10x more than necessary and takes years longer to complete).

Our payment gateway serves as the critical intermediary between merchant applications and external payment providers like Stripe, processing sensitive financial transactions that demand both functional correctness and robust operational characteristics. The scope decisions made here directly impact implementation complexity, security requirements, compliance obligations, and long-term maintainability.

Functional Goals

The functional goals define the core payment processing capabilities our system must deliver. Think of these as the fundamental promises our payment gateway makes to merchant applications that integrate with it. Each goal maps directly to one or more implementation milestones and represents functionality that must work reliably in production environments.

Payment Intent Management forms the foundation of our transaction processing model. A payment intent represents a customer's commitment to pay a specific amount for goods or services, created before any money actually moves. This abstraction separates the business decision to charge a customer from the technical complexity of processing that charge, enabling better error handling and user experience flows. Our system must create payment intents with client-provided idempotency keys, preventing duplicate charges when network issues cause request retries. The payment intent tracks metadata about the transaction (order number, customer ID, product details) and progresses through a well-defined state machine from creation to completion or cancellation. Intent expiration handling automatically cancels payment intents that remain uncompleted after reasonable time periods, preventing indefinite resource consumption.

Payment Intent Operation	Input Parameters	Expected Behavior	Error Conditions
Create Intent	amount, currency, idempotency_key, metadata	Returns intent ID and client secret	Invalid amount/currency, duplicate key with different params
Retrieve Intent	intent_id	Returns current intent state and metadata	Intent not found, unauthorized access
Cancel Intent	intent_id	Transitions to canceled state if possible	Intent already processed, invalid state transition
List Intents	filters, pagination	Returns matching intents with metadata	Invalid filter parameters, pagination limits

Payment Processing and Authentication handles the actual movement of money from customer payment methods to merchant accounts. Our system must tokenize payment method details (credit card numbers, bank account information) to avoid storing sensitive data directly. The charge creation process submits tokenized payment information to external providers and handles both immediate responses and asynchronous processing flows. 3D Secure authentication support ensures compliance with Strong Customer Authentication requirements by redirecting customers to their card issuer for identity verification when required. Payment confirmation completes the charge process after successful authentication, updating local records and triggering downstream business processes.

The key complexity here lies in handling the state transitions correctly. Unlike simple API calls, payment processing involves multiple round-trips between customer, merchant, payment provider, and card issuer. Our system must maintain consistent state throughout these interactions while gracefully handling failures at any step.

Payment Processing Stage	Required Actions	Success Criteria	Failure Handling
Method Tokenization	Securely exchange card details for token	Token stored, card details discarded	Invalid card data, provider rejection
Charge Creation	Submit payment request to provider	Charge ID returned, status tracking initiated	Network failure, provider error response
3DS Authentication	Handle issuer redirect flow	Customer authenticated, ready to capture	Authentication declined, timeout
Payment Confirmation	Complete authorized payment	Money transferred, local state updated	Capture failure, settlement delay

Refund and Dispute Management provides mechanisms for reversing completed payments and handling customer-initiated chargebacks. Full refund processing returns the entire payment amount to the customer's original payment method, while partial refund support enables returning portions of the original charge (useful for partial order cancellations or shipping adjustments). Refund state tracking monitors the refund lifecycle from initiation through completion, as refunds typically take 5-10 business days to appear on customer statements. Dispute handling manages chargeback notifications received from payment networks, tracking deadlines for evidence submission and automating responses where possible. Chargeback prevention features help identify potentially disputable transactions before they become formal disputes.

The accounting complexity in refund management requires careful attention to ledger integrity. Our system must track total refunded amounts against original charges to prevent over-refunding, maintain audit trails for compliance purposes, and handle the timing differences between refund initiation and actual fund return.

Refund Operation	Input Requirements	Processing Logic	State Updates
Full Refund	charge_id, reason	Validate refundable amount, submit to provider	Create refund record, update charge
Partial Refund	charge_id, amount, reason	Check remaining refundable balance	Track cumulative refund total
Dispute Response	dispute_id, evidence	Submit evidence within deadline	Update dispute status, set alerts
Chargeback Processing	webhook_data	Parse reason codes, determine response	Create dispute record, notify merchant

Webhook Processing and Event Handling maintains synchronization between our local payment state and the authoritative state held by external payment providers. Webhooks provide near-real-time notifications about payment status changes, refund completions, dispute updates, and other significant events. Our system must receive webhook payloads, verify their cryptographic signatures to prevent spoofing attacks, and update local records based on the event data. Event deduplication ensures that webhook retries don't cause duplicate processing, while event ordering handles cases where webhooks arrive out of chronological sequence.

Webhook reconciliation provides a safety net against webhook delivery failures by periodically comparing local payment state against provider APIs. This background process identifies discrepancies and corrects them, ensuring our local records remain consistent with provider records even when webhooks fail silently.

Non-Functional Goals

Non-functional goals define how well our payment system performs its functional duties. These quality attributes are just as critical as functional capabilities because payment processing demands high reliability, security, and performance standards. Think of non-functional goals as the operational excellence requirements that enable the system to handle real-world production workloads.

Security and Compliance Standards ensure our payment gateway meets industry requirements for handling sensitive financial data. PCI DSS compliance is mandatory for any system that processes, stores, or transmits payment card information. Our implementation targets `SAQ_A` compliance level by avoiding direct handling of card data, instead relying on tokenization to replace sensitive information with non-sensitive tokens. All data transmission must use `TLS_1_2` or higher encryption, and webhook signature verification prevents unauthorized parties from spoofing payment events. Access control mechanisms ensure only authenticated users can access payment information, with proper authorization checks preventing cross-customer data exposure.

Secure coding practices include input validation on all API endpoints, SQL injection prevention through parameterized queries, and proper error handling that doesn't leak sensitive information in responses. Audit logging captures all significant payment operations for compliance reporting and security incident investigation.

Security Requirement	Implementation Approach	Validation Method	Compliance Impact
PCI DSS Compliance	Tokenization, no card storage	Self-assessment questionnaire	Reduces compliance scope
Data Encryption	TLS 1.2+ for transport, encrypted storage	Certificate validation, encryption testing	Protects data in transit/rest
Access Control	Authentication + authorization	Security testing, access reviews	Prevents unauthorized access
Audit Logging	Structured logs for all operations	Log analysis, retention compliance	Supports compliance reporting

Reliability and Error Handling characteristics ensure payment processing continues functioning correctly despite various failure scenarios. Idempotency guarantees prevent duplicate charges when clients retry requests due to network issues or timeouts. Database transaction isolation ensures payment state changes are atomic—either all related updates succeed or none do, preventing partial state corruption. Circuit breaker patterns protect against cascading failures when external payment providers experience outages. Graceful degradation allows the system to continue operating with reduced functionality rather than complete failure.

Error recovery mechanisms handle both transient failures (network timeouts, temporary provider unavailability) and permanent failures (invalid payment methods, insufficient funds). Retry strategies use exponential backoff to avoid overwhelming failing systems while still providing timely recovery when issues resolve.

Reliability Feature	Failure Scenarios Addressed	Recovery Mechanism	Success Metrics
Idempotency	Network retries, client errors	Unique constraint on idempotency keys	Zero duplicate charges
Transaction Integrity	Database failures, partial updates	Atomic database transactions	Consistent state recovery
Circuit Breakers	Provider outages, cascading failures	Automatic failover, degraded mode	Maintained availability
Retry Logic	Transient network/provider issues	Exponential backoff with jitter	Successful eventual processing

Performance and Scalability requirements enable the system to handle realistic payment volumes with acceptable response times. Payment intent creation should complete within 200ms for 95% of requests, providing responsive user experience during checkout flows. Payment processing latency depends on external provider performance but should typically complete within 2-3 seconds for card payments not requiring 3D Secure authentication. Webhook processing must handle provider retry patterns without overwhelming the system, typically processing events within 1-2 seconds of receipt.

Throughput requirements scale with merchant transaction volumes. A typical small-to-medium merchant might process 100-1000 payments per day, while larger merchants could handle 10,000+ daily transactions. Our architecture should gracefully scale from development workloads to production volumes without requiring fundamental design changes.

Database performance optimization ensures payment queries remain fast as transaction history grows. Proper indexing on frequently-queried fields (payment intent ID, customer ID, creation timestamps) and archive strategies for historical data maintain query performance over time.

Performance Metric	Target Threshold	Measurement Method	Optimization Strategies
Intent Creation Latency	200ms p95	Application monitoring	Database indexing, caching
Payment Processing Time	3s p95 (non-3DS)	End-to-end tracing	Provider optimization, parallelization
Webhook Processing	2s p95	Queue monitoring	Background job processing
System Throughput	1000+ payments/day	Load testing	Horizontal scaling, async processing

Monitoring and Observability capabilities provide visibility into system health, performance trends, and business metrics. Application metrics track key performance indicators like payment success rates, average processing times, and error frequencies. Payment-specific dashboards display business-critical information such as daily transaction volumes, refund rates, and dispute trends. Alert systems notify operations teams about system degradation, unusual error patterns, or compliance-related issues.

Structured logging captures detailed information about each payment operation, enabling debugging of specific transaction issues and forensic analysis when problems occur. Distributed tracing follows payment requests across multiple system components and external provider calls, helping identify performance bottlenecks and failure points.

Explicit Non-Goals

Explicitly documenting what our payment gateway will NOT implement is crucial for maintaining reasonable scope and preventing feature creep. These non-goals represent important payment-related functionality that we consciously defer to future development phases or external systems. Think of non-goals as the features that would be nice to have but would significantly increase implementation complexity beyond the core learning objectives.

Advanced Fraud Detection and Risk Management represents a sophisticated domain that could easily consume months of development effort. While basic validation (amount limits, currency checks) falls within our scope, we explicitly exclude machine learning-based fraud scoring, velocity checks across customer behaviors, device fingerprinting, and advanced risk assessment algorithms. Payment providers typically offer fraud protection services that merchants can enable, reducing the need for custom fraud detection in many scenarios.

Real-world fraud detection requires extensive historical data, complex rule engines, and specialized expertise in payment fraud patterns. Building effective fraud prevention is a full-time specialty that extends far beyond basic payment processing implementation.

Fraud Detection Feature	Complexity Level	Why Excluded	Alternative Approach
ML Fraud Scoring	Very High	Requires data science expertise, training data	Use provider fraud tools
Velocity Checking	High	Complex rules, customer behavior tracking	Implement basic rate limiting only
Device Fingerprinting	High	Browser/device analysis, privacy concerns	Rely on provider capabilities
Risk Assessment Rules	Medium-High	Domain expertise, extensive configuration	Use provider risk management

Multi-Currency and International Support adds significant complexity around exchange rates, local payment methods, regulatory compliance, and tax calculation. While our system supports specifying currency codes for payments, we don't implement currency conversion, localized payment methods (SEPA, Alipay, etc.), country-specific compliance requirements, or multi-currency settlement. International payment processing involves understanding banking regulations, tax obligations, and business practices across multiple jurisdictions.

Supporting international payments properly requires partnerships with local payment providers, understanding of regional customer preferences, and compliance with diverse regulatory frameworks. This expansion would multiply the complexity of every system component.

Subscription and Recurring Billing involves complex business logic around billing cycles, proration calculations, dunning management for failed payments, and subscription lifecycle management. While our payment processing engine could theoretically handle recurring charges, we exclude the subscription management layer that would coordinate billing schedules, handle plan changes, manage customer communication for billing issues, and integrate with business systems for usage-based billing.

Subscription billing is essentially a complete business application built on top of payment processing primitives. The complexity lies not in charging cards repeatedly, but in managing all the business rules, customer communication, and operational processes around subscription relationships.

Subscription Feature	Business Complexity	Technical Complexity	External Dependencies
Billing Cycles	High - multiple cycle types, proration	Medium - scheduling, date math	Calendar systems, time zones
Dunning Management	High - customer communication, retry logic	Medium - workflow automation	Email systems, notification services
Plan Management	Very High - pricing rules, feature gates	High - configuration management	Product catalog, entitlement systems
Usage Billing	Very High - metering, rating, aggregation	Very High - data processing pipelines	Usage tracking, analytics systems

Marketplace and Multi-Party Payments enable platforms to facilitate payments between multiple parties with split settlements, escrow functionality, and complex fee structures. While conceptually similar to basic payment processing, marketplace payments require additional legal frameworks, compliance considerations, and business logic around fund holding, dispute resolution between multiple parties, and tax reporting for platform transactions.

The regulatory complexity of marketplace payments varies significantly by jurisdiction, often requiring money transmitter licenses and additional compliance obligations beyond basic payment processing.

Advanced Reporting and Analytics beyond basic transaction reporting falls outside our scope. We exclude comprehensive business intelligence dashboards, predictive analytics, customer behavior analysis, and advanced financial reporting. While our system maintains audit logs and basic transaction history, we don't build sophisticated reporting tools, data warehousing capabilities, or integration with business intelligence platforms.

Payment analytics can become extremely sophisticated, involving customer segmentation, cohort analysis, revenue forecasting, and integration with broader business metrics. This represents a separate application domain that consumes payment data rather than processing payments directly.

Design Principle: Focused Scope

By explicitly limiting scope to core payment processing functionality, we can implement a robust, well-tested payment gateway within reasonable time constraints. Each excluded feature could easily double the implementation effort and introduce additional failure modes, external dependencies, and compliance requirements. The goal is building a solid foundation that supports future extensions rather than attempting to solve every payment-related problem immediately.

Legacy System Integration with existing merchant systems, ERP platforms, accounting software, and custom business applications requires extensive configuration, data transformation, and workflow coordination capabilities. While our payment gateway provides standard REST APIs that can integrate with various systems, we don't build custom connectors, data synchronization tools, or workflow orchestration engines for specific business systems.

Integration complexity grows exponentially with the number and diversity of systems involved. Each integration often requires understanding the target system's data model, business processes, and technical constraints—work that typically requires business analysis and custom development for each merchant.

Compliance Beyond PCI DSS such as SOX financial reporting, GDPR data protection, industry-specific regulations (healthcare, government), and regional payment regulations represents additional complexity that extends beyond basic

payment processing. While our system design considers privacy and security principles, we don't implement comprehensive compliance frameworks for specialized industries or jurisdictions.

Different compliance requirements often conflict with each other or require extensive documentation, testing, and audit procedures that significantly impact development timelines and system complexity.

Implementation Guidance

Building a payment gateway with clear scope boundaries requires establishing development practices that resist feature creep while ensuring core functionality meets production quality standards. The technology choices and project structure should support focused implementation of the defined goals while making future extensions possible.

A. Technology Recommendations

Component	Simple Option	Advanced Option
Web Framework	Flask with minimal extensions	FastAPI with automatic OpenAPI generation
Database	SQLite for development, PostgreSQL production	PostgreSQL with read replicas and connection pooling
Background Jobs	Simple cron scripts for reconciliation	Celery with Redis for webhook processing
Monitoring	Python logging with structured JSON	APM tools like DataDog or New Relic
Testing	pytest with requests-mock	pytest with Docker containers for integration tests
Configuration	Environment variables with python-decouple	HashiCorp Consul or similar config management
Authentication	Simple API keys	JWT tokens with proper key rotation
Documentation	Markdown files in repository	Generated API docs with examples

B. Recommended Project Structure

```

payment-gateway/
├── app/
│   ├── __init__.py
│   ├── main.py          # FastAPI application setup
│   ├── config.py        # Configuration management
│   ├── database.py      # Database connection and session management
│   ├── models/
│   │   ├── __init__.py
│   │   ├── payment_intent.py    # PaymentIntent data model
│   │   ├── charge.py          # Charge data model
│   │   ├── refund.py          # Refund data model
│   │   └── webhook_event.py   # WebhookEvent data model
│   ├── services/
│   │   ├── __init__.py
│   │   ├── payment_intent_service.py  # Business logic for payment intents
│   │   ├── payment_processor.py      # External provider integration
│   │   ├── refund_service.py        # Refund processing logic
│   │   └── webhook_service.py      # Webhook processing and verification
│   ├── api/
│   │   ├── __init__.py
│   │   ├── payment_intents.py      # Payment intent API endpoints
│   │   ├── charges.py            # Charge processing endpoints
│   │   ├── refunds.py            # Refund management endpoints
│   │   └── webhooks.py           # Webhook reception endpoints
│   ├── utils/
│   │   ├── __init__.py
│   │   ├── validation.py         # Input validation functions
│   │   ├── crypto.py            # Signature verification utilities
│   │   └── exceptions.py        # Custom exception classes
│   └── background/
│       ├── __init__.py
│       ├── reconciliation.py    # State reconciliation jobs
│       └── cleanup.py          # Intent expiration cleanup
└── migrations/
└── tests/
    ├── unit/                 # Unit tests for individual components
    ├── integration/          # Integration tests with mocked providers
    └── fixtures/             # Test data and mock responses
└── docker-compose.yml      # Development environment setup
└── requirements.txt        # Python dependencies
└── .env.example             # Environment variable template
└── README.md               # Setup and development instructions

```

C. Infrastructure Starter Code

Here's complete infrastructure code for validation utilities that support the defined goals:

```
# app/utils/validation.py                                         PYTHON

"""
Validation utilities for payment gateway operations.

Provides functions for validating monetary amounts, currency codes, and idempotency keys.

"""

import re

from decimal import Decimal, InvalidOperation

from typing import Dict, Any

from app.utils.exceptions import ValidationError

# ISO 4217 currency codes commonly supported by payment providers

SUPPORTED_CURRENCIES = {

    'USD', 'EUR', 'GBP', 'CAD', 'AUD', 'JPY', 'CHF', 'CNY', 'SGD', 'HKD'
}

# Idempotency key format: alphanumeric with hyphens, 1-64 characters

IDEMPOTENCY_KEY_PATTERN = re.compile(r'^[a-zA-Z0-9\-\-]{1,64}$')

def validate_amount(amount: Any) -> int:

    """
    Validates and converts monetary amount to cents.

    Accepts string, int, float, or Decimal inputs representing dollar amounts.

    Returns integer cents to avoid floating point precision issues.

    Args:
        amount: Monetary amount in dollars (e.g., "10.50", 10.5, 1050 cents)

    Returns:
        int: Amount in cents (e.g., 1050 for $10.50)
    """

    if isinstance(amount, str):
        amount = Decimal(amount.replace(',', '.'))
    elif isinstance(amount, float):
        amount = Decimal(str(amount))
    elif isinstance(amount, int):
        amount = Decimal(str(amount))

    if amount < 0:
        raise InvalidOperation("Amount must be non-negative")

    return int(amount * 100)
```

```
Raises:  
    ValidationError: If amount is invalid, negative, or exceeds limits  
  
"""  
  
if amount is None:  
    raise ValidationError("Amount cannot be None")  
  
  
try:  
  
    # Handle cents input (integers >= 50 assumed to be cents already)  
  
    if isinstance(amount, int):  
  
        if amount >= 50: # Assume cents for amounts >= 50  
  
            cents = amount  
  
        else:  
  
            cents = amount * 100 # Convert dollars to cents  
  
    else:  
  
        # Handle string/float/Decimal dollar amounts  
  
        decimal_amount = Decimal(str(amount))  
  
        cents = int(decimal_amount * 100)  
  
except (ValueError, InvalidOperation, OverflowError):  
    raise ValidationError(f"Invalid amount format: {amount}")  
  
  
# Validate amount constraints  
  
if cents < 0:  
    raise ValidationError("Amount cannot be negative")  
  
if cents == 0:  
    raise ValidationError("Amount must be greater than zero")  
  
if cents > 99999999: # $999,999.99 maximum  
    raise ValidationError("Amount exceeds maximum allowed value")  
  
  
return cents
```

```
def validate_currency(currency: str) -> str:

    """
    Validates ISO 4217 currency code.

    Args:
        currency: Three-letter currency code (e.g., "USD", "EUR")

    Returns:
        str: Uppercase currency code

    Raises:
        ValidationError: If currency is invalid or unsupported

    """
    if not currency or not isinstance(currency, str):
        raise ValidationError("Currency code is required")

    currency_upper = currency.upper().strip()

    if len(currency_upper) != 3:
        raise ValidationError("Currency code must be exactly 3 characters")

    if not currency_upper.isalpha():
        raise ValidationError("Currency code must contain only letters")

    if currency_upper not in SUPPORTED_CURRENCIES:
        raise ValidationError(f"Unsupported currency: {currency_upper}")

    return currency_upper

def validate_idempotency_key(key: str) -> str:
```

```
"""
    Validates idempotency key format and uniqueness requirements.
```

Args:

```
    key: Client-provided idempotency key
```

Returns:

```
    str: Validated idempotency key
```

Raises:

```
    ValidationError: If key format is invalid
```

```
"""

if not key or not isinstance(key, str):
```

```
    raise ValidationError("Idempotency key is required")
```

```
key_stripped = key.strip()
```

```
if not IDEMPOTENCY_KEY_PATTERN.match(key_stripped):
```

```
    raise ValidationError(
```

```
        "Idempotency key must be 1-64 characters, alphanumeric with hyphens only"
```

```
)
```

```
return key_stripped
```

```
def validate_metadata(metadata: Dict[str, Any]) -> Dict[str, str]:
```

```
"""

Validates and normalizes payment intent metadata.
```

Args:

```
    metadata: Dictionary of key-value pairs for payment context
```

Returns:

```
Dict[str, str]: Validated metadata with string values
```

Raises:

```
ValidationError: If metadata format is invalid
```

```
"""
```

```
if metadata is None:
```

```
    return {}
```

```
if not isinstance(metadata, dict):
```

```
    raise ValidationError("Metadata must be a dictionary")
```

```
if len(metadata) > 20:
```

```
    raise ValidationError("Metadata cannot have more than 20 keys")
```

```
validated = []
```

```
for key, value in metadata.items():
```

```
    if not isinstance(key, str):
```

```
        raise ValidationError("Metadata keys must be strings")
```

```
    if len(key) > 40:
```

```
        raise ValidationError("Metadata keys cannot exceed 40 characters")
```

```
# Convert values to strings for storage
```

```
str_value = str(value) if value is not None else ""
```

```
if len(str_value) > 500:
```

```
    raise ValidationError("Metadata values cannot exceed 500 characters")
```

```
validated[key] = str_value
```

```
return validated
```

```
# app/utils/exceptions.py

"""
Custom exception classes for payment gateway operations.

Provides structured error handling with appropriate HTTP status codes.

"""

from typing import Optional, Dict, Any

class PaymentGatewayException(Exception):

    """Base exception for all payment gateway errors."""

    def __init__(self, message: str, error_code: str = "UNKNOWN_ERROR",
                 details: Optional[Dict[str, Any]] = None):
        super().__init__(message)

        self.message = message
        self.error_code = error_code
        self.details = details or {}

class ValidationError(PaymentGatewayException):

    """Raised when input validation fails."""

    def __init__(self, message: str, field: Optional[str] = None):
        super().__init__(message, "VALIDATION_ERROR", {"field": field})
        self.field = field

class IdempotencyError(PaymentGatewayException):

    """Raised when idempotency key conflicts occur."""

    def __init__(self, message: str, existing_intent_id: Optional[str] = None):
        super().__init__(message, "IDEMPOTENCY_ERROR",
                        {"existing_intent_id": existing_intent_id})
```

PYTHON

```
class PaymentProcessingError(PaymentGatewayException):

    """"Raised when payment processing fails.""""

    def __init__(self, message: str, provider_error: Optional[str] = None,
                 retry_allowed: bool = False):
        super().__init__(message, "PAYMENT_ERROR", {
            "provider_error": provider_error,
            "retry_allowed": retry_allowed
        })

class WebhookVerificationError(PaymentGatewayException):

    """"Raised when webhook signature verification fails.""""

    def __init__(self, message: str = "Webhook signature verification failed"):
        super().__init__(message, "WEBHOOK_VERIFICATION_ERROR")

class StateTransitionError(PaymentGatewayException):

    """"Raised when invalid state transitions are attempted.""""

    def __init__(self, message: str, current_state: str, attempted_state: str):
        super().__init__(message, "STATE_TRANSITION_ERROR", {
            "current_state": current_state,
            "attempted_state": attempted_state
        })
```

D. Core Logic Skeleton

For the goal validation and scoping logic that learners should implement:

```
# app/services/scope_validator.py
```

PYTHON

```
"""  
  
Service for validating payment operations against defined system goals.  
  
Ensures requests fall within functional scope and meet non-functional requirements.  
"""
```

```
from typing import Dict, Any, List  
  
from app.utils.exceptions import ValidationError, PaymentGatewayException
```

```
class ScopeValidator:
```

```
    """Validates payment operations against system scope boundaries."""
```

```
    def validate_payment_request(self, request_data: Dict[str, Any]) -> None:
```

```
        """  
  
        Validates payment request against functional goals.  
  
        Args:  
            request_data: Payment request parameters  
  
        Raises:  
            ValidationError: If request exceeds defined scope  
  
        """
```

```
# TODO 1: Validate amount is within supported range (avoid complex currency conversion)  
  
# TODO 2: Check currency is in SUPPORTED_CURRENCIES (no international complexity)  
  
# TODO 3: Verify payment method type is supported (cards only, no exotic methods)  
  
# TODO 4: Ensure metadata doesn't contain fraud-detection related fields  
  
# TODO 5: Check request doesn't include subscription/recurring parameters  
  
# Hint: Use validate_amount, validate_currency from validation.py  
  
pass
```

```
    def check_performance_requirements(self, operation_type: str,
```

```
        start_time: float, current_time: float) -> None:

"""

Validates operation meets performance goals.

Args:

    operation_type: Type of operation (intent_creation, payment_processing, etc.)
    start_time: Operation start timestamp
    current_time: Current timestamp

Raises:

    PaymentGatewayException: If performance goals are not met

"""

# TODO 1: Calculate elapsed time from timestamps

# TODO 2: Define performance thresholds for each operation type

# TODO 3: Log warning if approaching threshold (80% of limit)

# TODO 4: Raise exception if threshold exceeded

# TODO 5: Update performance metrics for monitoring

# Hint: intent_creation should be < 200ms, payment_processing < 3s

pass


def validate_compliance_requirements(self, data_fields: List[str]) -> None:

"""

Ensures request meets PCI DSS compliance requirements.

Args:

    data_fields: List of field names in the request

Raises:

    ValidationError: If request contains non-compliant data

"""
```

```

# TODO 1: Check for prohibited fields (card_number, cvv, etc.)

# TODO 2: Ensure only tokenized payment methods are used

# TODO 3: Validate TLS encryption is enforced

# TODO 4: Check audit logging requirements are met

# TODO 5: Verify access control checks are in place

# Hint: Any field containing raw card data should trigger error

pass

```

E. Language-Specific Hints

For Python payment gateway development:

- Use `pydantic` models for request/response validation with automatic OpenAPI generation
- Implement database models with SQLAlchemy ORM for proper relationship handling
- Use `python-decimal` for all monetary calculations to avoid floating-point precision issues
- Store all amounts as integers in cents to prevent rounding errors
- Use `secrets` module for generating secure idempotency keys and API tokens
- Implement proper exception handling with custom exception hierarchy
- Use `structlog` for structured logging with correlation IDs for request tracing
- Apply `dataclass` for internal data transfer objects to ensure type safety

F. Milestone Checkpoints

After implementing goal validation and scope boundaries:

Checkpoint 1: Functional Goal Validation

- Run: `python -m pytest tests/unit/test_scope_validator.py -v`
- Expected: All validation functions properly reject out-of-scope requests
- Manual test: Send payment request with unsupported currency, should return `ValidationError`
- Manual test: Include subscription parameters, should be rejected
- Signs of issues: Validator accepts requests that exceed defined scope

Checkpoint 2: Performance Goal Monitoring

- Run: `curl -X POST localhost:8000/payment_intents` and check response time
- Expected: Response headers include timing information, sub-200ms for intent creation
- Manual test: Create payment intent, verify response time meets goals
- Signs of issues: Response times consistently exceed defined thresholds

Checkpoint 3: Compliance Boundary Enforcement

- Run: `python -m pytest tests/unit/test_compliance.py -v`
- Expected: Any request containing raw card data is rejected
- Manual test: Submit request with `card_number` field, should fail validation

- Signs of issues: System accepts or processes prohibited data fields

High-Level Architecture

Milestone(s): Foundation for all milestones - establishes the system structure that supports payment intents (Milestone 1), payment processing (Milestone 2), refunds and disputes (Milestone 3), and webhook reconciliation (Milestone 4)

Think of our payment gateway as a sophisticated financial relay station, similar to how a major airport manages thousands of flight connections daily. Just as an airport has distinct terminals (domestic, international, cargo), specialized staff (air traffic control, security, customs), and coordination systems (flight tracking, baggage handling), our payment gateway orchestrates multiple specialized components that must work in perfect harmony to move money safely from customers to merchants.

The architectural challenge mirrors managing a complex transportation hub: we must handle high-volume traffic (thousands of payments per minute), maintain strict security protocols (PCI compliance), coordinate with external systems (payment providers, banks), and ensure nothing gets lost in transit (payment reconciliation). Each component has specialized responsibilities, but they must communicate seamlessly to deliver a unified payment experience.

Component Overview

Our payment gateway architecture divides responsibilities among five core components, each designed with a specific purpose in the overall payment processing ecosystem. This separation follows the principle of single responsibility while enabling loose coupling between components for maintainability and scalability.

Payment Intent Manager serves as the initial point of contact for all payment requests, functioning like a reservation system at a restaurant. When a customer decides to make a purchase, this component creates a payment intent - essentially a commitment that payment will be attempted with specific terms (amount, currency, metadata). The manager handles idempotency key validation to prevent duplicate charges, manages the payment intent lifecycle through various states, and provides the foundation for all subsequent payment operations.

Processing Engine acts as the core transaction processor, similar to how a bank's wire transfer department handles actual money movement. This component takes validated payment intents and orchestrates the complex process of charging payment methods through external providers. It manages payment method tokenization for security, implements 3D Secure authentication flows for regulatory compliance, and handles the intricate dance of authorization and capture that defines modern payment processing.

Refund and Dispute Handler manages the post-payment lifecycle, functioning like a customer service department with the authority to reverse transactions. This component processes both merchant-initiated refunds and customer-initiated chargebacks, maintains careful accounting to ensure refund amounts never exceed original charges, and implements the complex workflows required for dispute evidence submission and chargeback management.

Webhook Processor serves as the system's nervous system, receiving and processing real-time notifications from external payment providers. Like a financial news ticker that updates traders on market movements, this component ensures our local payment state stays synchronized with the authoritative state maintained by payment providers, handling the complexities of signature verification, event deduplication, and ordered processing.

State Reconciliation Service acts as the system's auditing department, periodically comparing our local payment records against the authoritative state maintained by payment providers. This component catches discrepancies that might arise from missed webhooks, network failures, or processing errors, ensuring long-term data consistency even in the face of temporary communication failures.

The following table details each component's specific responsibilities:

Component	Primary Responsibility	Key Operations	Data Ownership
Payment Intent Manager	Intent lifecycle management	Create intent, validate idempotency, track expiration	PaymentIntent records, idempotency keys
Processing Engine	Payment execution	Tokenize payment methods, process charges, handle 3DS	Charge records, payment method tokens
Refund and Dispute Handler	Payment reversals	Process refunds, handle chargebacks, manage disputes	Refund records, dispute evidence
Webhook Processor	Real-time state sync	Verify signatures, process events, update records	WebhookEvent logs, processing state
State Reconciliation Service	Periodic consistency checks	Compare states, detect discrepancies, trigger corrections	Reconciliation reports, sync status

Decision: Microservice vs Monolithic Architecture

- **Context:** Payment systems require high reliability and clear component boundaries, but also need strong consistency guarantees and transactional integrity
- **Options Considered:**
 1. Microservices with event-driven communication
 2. Modular monolith with clear internal boundaries
 3. Hybrid approach with core payments as monolith and auxiliary services separate
- **Decision:** Modular monolith with clear internal component boundaries
- **Rationale:** Payment processing requires ACID transactions across multiple operations (intent creation, charge processing, state updates). Distributed transactions are complex and introduce failure modes. A monolith allows database transactions to span components while still maintaining clear separation of concerns through interfaces and dependency injection.
- **Consequences:** Simpler deployment and testing, stronger consistency guarantees, but requires discipline to maintain component boundaries and may require refactoring to microservices if scaling demands exceed single-instance capabilities.

Data Flow Patterns

Payment data flows through our system following predictable patterns that mirror real-world financial transactions. Understanding these flows is crucial because payment processing is inherently stateful - each step depends on the successful completion of previous steps, and failures at any point require careful rollback or compensation logic.

Intent Creation Flow begins when a client application (web frontend, mobile app, or API integration) initiates a payment request. The flow starts with HTTP request validation, ensuring all required fields are present and properly formatted. The system validates the monetary amount using `validate_amount()` to convert dollar amounts to cents and prevent floating-point arithmetic errors. Currency validation through `validate_currency()` ensures compliance with ISO 4217 standards. The idempotency key undergoes validation via `validate_idempotency_key()` to ensure proper format and uniqueness constraints.

The Payment Intent Manager checks for existing intents with the same idempotency key. If found, it returns the existing intent without creating a duplicate, implementing true idempotent behavior. If no duplicate exists, the manager creates a new `PaymentIntent` record with status `created`, stores it in the database within a transaction, and returns the intent details including a client secret for subsequent operations.

Payment Processing Flow activates when a client confirms a payment intent by submitting payment method details. The Processing Engine receives the confirmation request along with the payment method information (credit card details, bank account information, or alternative payment method tokens). The engine immediately tokenizes sensitive payment data, replacing credit card numbers with opaque tokens that cannot be reverse-engineered to reveal the original information.

For credit card payments requiring 3D Secure authentication, the engine initiates the authentication challenge by redirecting the customer to their issuing bank's authentication page. The customer completes the authentication (entering a password, using biometric verification, or confirming via mobile app), and the bank redirects back to our system with authentication results.

Upon successful authentication (or for payments not requiring 3DS), the engine submits the charge request to the external payment provider. The provider processes the payment through banking networks, applying fraud checks, verifying account balances, and obtaining authorization from the issuing bank. The response updates our local `Charge` record with the final status and transaction identifiers.

Webhook Reconciliation Flow operates asynchronously as payment providers send real-time notifications about payment state changes. The Webhook Processor receives HTTP POST requests at our webhook endpoint, immediately verifying the payload signature using HMAC-SHA256 to ensure authenticity and prevent spoofing attacks.

Valid webhooks undergo event processing where the processor extracts payment identifiers, determines the event type (charge succeeded, payment failed, refund completed), and updates corresponding local records. The processor implements deduplication by checking webhook event IDs against previously processed events, ensuring idempotent handling even when providers retry webhook delivery.

State Reconciliation Flow runs periodically (typically every 15-30 minutes) to catch any discrepancies between local payment state and provider authoritative state. The reconciliation service queries recent payment records from our database, fetches corresponding payment status from external providers via API calls, and compares the states. Discrepancies trigger alert notifications and automated correction workflows where possible.

The following table illustrates the complete data transformation journey:

Flow Stage	Input Data	Processing Component	Output Data	Persistence Layer
Intent Creation	Client payment request	Payment Intent Manager	PaymentIntent with status <code>created</code>	PostgreSQL payment_intents table
Method Tokenization	Raw payment method data	Processing Engine	Payment method token	Secure token vault (encrypted)
3DS Authentication	Customer credentials	External issuer bank	Authentication result token	Processing Engine memory (temporary)
Charge Processing	Tokenized payment method	Processing Engine	Charge with status succeeded / failed	PostgreSQL charges table
Webhook Processing	Provider notification	Webhook Processor	Updated payment status	PostgreSQL webhook_events table
State Reconciliation	Local vs provider state	Reconciliation Service	Consistency report	PostgreSQL reconciliation_logs table

The critical insight here is that payment data never travels in a straight line - it flows through multiple validation, transformation, and storage steps, with each step designed to be recoverable in case of failures. This pipeline approach ensures we can trace any payment's journey and resume processing from any interruption point.

Error Recovery Flow activates when any step in the normal processing flow encounters a failure. The system maintains detailed audit logs at each processing stage, allowing precise identification of failure points. For transient failures (network timeouts, provider rate limiting), the system implements exponential backoff retry logic with jitter to prevent thundering herd problems. For permanent failures (invalid payment methods, insufficient funds), the system updates payment status accordingly and triggers customer notification workflows.

Recommended Project Structure

A well-organized codebase is essential for payment systems because regulatory compliance, security audits, and maintenance require clear separation between different types of functionality. Our project structure reflects the architectural components while providing clear boundaries between business logic, external integrations, and infrastructure concerns.

The project follows a layered architecture pattern where each directory has a specific purpose and dependency direction flows from outer layers (HTTP handlers) toward inner layers (domain logic) toward infrastructure layers (database, external APIs). This structure enables easy testing by allowing infrastructure dependencies to be mocked or stubbed during unit tests.

Top-Level Organization separates executable entry points, internal business logic, external integrations, database schemas, and configuration. The `cmd/` directory contains main functions for different deployment modes (web server, background workers, CLI tools). The `internal/` directory houses all business logic and is not importable by external packages, enforcing encapsulation. The `pkg/` directory contains utilities that could be shared with other projects. The `migrations/` directory contains database schema evolution scripts with version control.

Component Module Structure within the `internal/` directory maps directly to our architectural components. Each component gets its own subdirectory with a consistent internal structure: main business logic files, test files, internal

interfaces, and component-specific utilities. This organization makes it easy for developers to locate functionality and maintain clear boundaries between components.

External Integration Structure isolates all third-party dependencies in dedicated modules. Payment provider integrations live in separate subdirectories under `internal/providers/`, allowing easy swapping between different providers (Stripe, PayPal, Adyen) through common interfaces. Database access layers provide abstraction over SQL queries, enabling database migrations and testing with different database engines.

Configuration and Deployment Structure separates environment-specific configuration from code. The `config/` directory contains YAML or JSON files for different deployment environments (development, staging, production). Docker files and Kubernetes manifests live in `deployments/` directory. Documentation and API specifications reside in `docs/` directory.

Here's the complete recommended project structure:

```
payment-gateway/
├── cmd/
│   ├── server/
│   │   └── main.go          # HTTP server entry point
│   ├── worker/
│   │   └── main.go          # Background job processor
│   └── reconciler/
│       └── main.go          # State reconciliation service
├── internal/
│   ├── intent/
│   │   ├── manager.go        # PaymentIntent business logic
│   │   ├── manager_test.go    # Intent manager tests
│   │   ├── idempotency.go     # Idempotency key handling
│   │   └── expiration.go      # Intent expiration logic
│   ├── processing/
│   │   ├── engine.go          # Charge processing logic
│   │   ├── engine_test.go      # Processing engine tests
│   │   ├── three_ds.go         # 3D Secure authentication
│   │   └── tokenization.go     # Payment method tokenization
│   ├── refunds/
│   │   ├── handler.go         # Refund processing logic
│   │   ├── disputes.go        # Chargeback and dispute handling
│   │   └── accounting.go       # Payment accounting ledger
│   ├── webhooks/
│   │   ├── processor.go        # Webhook event processing
│   │   ├── verification.go      # Signature verification
│   │   └── deduplication.go     # Event deduplication
│   ├── reconciliation/
│   │   ├── service.go          # State reconciliation logic
│   │   └── scheduler.go        # Periodic reconciliation jobs
│   ├── providers/
│   │   ├── stripe/
│   │   │   ├── client.go        # Stripe API integration
│   │   │   └── webhooks.go       # Stripe webhook handling
│   │   └── interface.go        # Common provider interface
│   ├── storage/
│   │   ├── database.go         # Database connection management
│   │   ├── payment_intents.go    # PaymentIntent repository
│   │   ├── charges.go          # Charge repository
│   │   ├── refunds.go          # Refund repository
│   │   └── webhook_events.go     # WebhookEvent repository
│   ├── models/
│   │   ├── payment_intent.go    # PaymentIntent domain model
│   │   ├── charge.go           # Charge domain model
│   │   ├── refund.go            # Refund domain model
│   │   └── webhook_event.go      # WebhookEvent domain model
│   ├── validation/
│   │   ├── amount.go           # validate_amount implementation
│   │   ├── currency.go          # validate_currency implementation
│   │   └── idempotency.go        # validate_idempotency_key implementation
└── transport/
    ├── http/
    │   ├── handlers/
    │   │   ├── payment_intents.go # Payment intent HTTP handlers
    │   │   ├── charges.go          # Charge HTTP handlers
    │   │   ├── refunds.go          # Refund HTTP handlers
    │   │   └── webhooks.go         # Webhook HTTP handlers
    │   └── middleware/
    │       └── auth.go            # Authentication middleware
```

```

    |   |   |   └ rate_limit.go      # Rate limiting middleware
    |   |   └ logging.go          # Request logging middleware
    |   └ server.go              # HTTP server setup
    └ grpc/
        └ server.go              # gRPC server (future extension)
        └ proto/                  # Protocol buffer definitions

```

```

└ pkg/
    └ crypto/
        └ hmac.go                # HMAC signature utilities
        └ tokens.go               # Token generation utilities
    └ money/
        └ currency.go             # Currency handling utilities
        └ amounts.go               # Money amount utilities

```

```

└ migrations/
    └ 001_create_payment_intents.sql # Database schema migration
    └ 002_create_charges.sql        # Charge table creation
    └ 003_create_refunds.sql        # Refund table creation
    └ 004_create_webhook_events.sql # WebhookEvent table creation

```

```

└ config/
    └ development.yaml           # Development environment config
    └ staging.yaml               # Staging environment config
    └ production.yaml            # Production environment config

```

```

└ deployments/
    └ docker/
        └ Dockerfile               # Container image definition
    └ kubernetes/
        └ deployment.yaml           # Kubernetes deployment
        └ service.yaml              # Kubernetes service

```

```

└ docs/
    └ api/
        └ openapi.yaml              # API specification
    └ architecture/
        └ diagrams/                 # Architecture diagrams

```

```

└ scripts/
    └ setup-dev.sh               # Development environment setup
    └ run-tests.sh                # Test execution script

```

Decision: Repository Pattern vs Active Record

- **Context:** Payment data requires complex querying, transaction management, and needs to support multiple database engines for different deployment environments
- **Options Considered:**
 1. Active Record pattern with models containing database logic
 2. Repository pattern with separate data access layer
 3. Raw SQL with query builders
- **Decision:** Repository pattern with dedicated storage layer
- **Rationale:** Payment systems require complex transactional integrity across multiple tables. Repository pattern allows easy testing with mock implementations, supports multiple database backends, and keeps business logic separate from data persistence concerns. This is critical for PCI compliance audits which require clear separation of concerns.
- **Consequences:** More code to write and maintain, but better testability, cleaner business logic, and easier compliance with security standards.

Development Workflow Organization supports both individual developer productivity and team collaboration. Each component has its own test suite that can run independently, enabling fast feedback loops during development. Integration tests that span multiple components live in a separate `tests/` directory with realistic test data and scenarios.

The configuration structure supports environment-specific overrides while keeping sensitive data (API keys, database passwords) in environment variables rather than committed files. Database migrations are versioned and can be applied automatically during deployment, ensuring schema consistency across environments.

This structure scales from initial development through production deployment, supporting team growth by providing clear ownership boundaries and enabling parallel development across different components.

Implementation Guidance

The payment gateway architecture requires careful technology choices and a robust development foundation. This guidance provides concrete recommendations for implementing each architectural layer, with working starter code and clear development milestones.

A. Technology Recommendations Table:

Component	Simple Option	Advanced Option
Web Framework	FastAPI with SQLAlchemy	Django with Django REST Framework
Database	PostgreSQL with psycopg2	PostgreSQL with async SQLAlchemy
Queue System	Redis with RQ	Celery with Redis/RabbitMQ
Caching	Redis simple caching	Redis with clustering
HTTP Client	requests library	httpx with async support
Configuration	python-dotenv + YAML	Pydantic settings management
Logging	Python logging module	Structured logging with loguru
Testing	pytest with pytest-mock	pytest with pytest-asyncio
Monitoring	Basic logging to files	Prometheus + Grafana
Deployment	Docker with docker-compose	Kubernetes with Helm charts

B. Recommended File/Module Structure:

Here's how the payment gateway components map to Python modules:

```
payment_gateway/                                PYTHON

|__ main.py                                     # FastAPI application entry point

|__ config/
|  |__ __init__.py
|  |__ settings.py                            # Pydantic settings management
|  |__ database.py                           # Database configuration

|__ models/
|  |__ __init__.py
|  |__ payment_intent.py                     # PaymentIntent SQLAlchemy model
|  |__ charge.py                            # Charge SQLAlchemy model
|  |__ refund.py                            # Refund SQLAlchemy model
|  |__ webhook_event.py                    # WebhookEvent SQLAlchemy model

|__ schemas/
|  |__ __init__.py
|  |__ payment_intent.py                  # Pydantic schemas for API serialization
|  |__ charge.py                         # Charge request/response schemas
|  |__ refund.py                         # Refund request/response schemas

|__ services/
|  |__ __init__.py
|  |__ payment_intent_manager.py        # Intent lifecycle management
|  |__ processing_engine.py            # Payment processing logic
|  |__ refund_handler.py              # Refund and dispute management
|  |__ webhook_processor.py          # Webhook processing logic
|  |__ reconciliation_service.py    # State reconciliation

|__ repositories/
|  |__ __init__.py
|  |__ payment_intent_repository.py  # PaymentIntent data access
|  |__ charge_repository.py         # Charge data access
|  |__ webhook_event_repository.py # WebhookEvent data access

|__ providers/
```

C. Infrastructure Starter Code (COMPLETE, ready to use):

Here's complete database configuration and connection management:

```
# config/database.py

from sqlalchemy import create_engine, MetaData

from sqlalchemy.ext.declarative import declarative_base

from sqlalchemy.orm import sessionmaker

from sqlalchemy.pool import StaticPool

import os

DATABASE_URL = os.getenv(
    "DATABASE_URL",
    "postgresql://user:password@localhost/payment_gateway"
)

engine = create_engine(
    DATABASE_URL,
    poolclass=StaticPool,
    pool_pre_ping=True,
    echo=os.getenv("SQL_DEBUG", "false").lower() == "true"
)

SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

Base = declarative_base()

def get_db():
    """Database session dependency for FastAPI dependency injection."""
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()

def create_tables():
    """Create all database tables. Call this on application startup."""
```

```
Base.metadata.create_all(bind=engine)
```

Complete validation utilities implementation:

```
# utils/validation.py

import re

from typing import Union

from decimal import Decimal, InvalidOperation

from .exceptions import ValidationError

# ISO 4217 currency codes (subset for common currencies)

VALID_CURRENCIES = {

    'USD', 'EUR', 'GBP', 'JPY', 'CAD', 'AUD', 'CHF', 'CNY', 'SEK', 'NZD'

}
```

```
def validate_amount(amount: Union[str, int, float, Decimal]) -> int:
```

```
"""
```

```
    Converts monetary amount to cents with validation.
```

```
    Prevents floating point arithmetic errors by working in integer cents.
```

Args:

```
    amount: Amount in dollars (e.g., 10.50, "10.50", 1050 cents)
```

Returns:

```
    int: Amount in cents (e.g., 1050 for $10.50)
```

Raises:

```
    ValidationError: If amount is invalid, negative, or too large
```

```
"""
```

```
try:
```

```
    if isinstance(amount, str):
        decimal_amount = Decimal(amount)
    else:
        decimal_amount = Decimal(str(amount))
```

```
# Convert to cents

cents = int(decimal_amount * 100)

if cents < 0:

    raise ValidationError("Amount cannot be negative")

if cents > 99999999: # $999,999.99 maximum

    raise ValidationError("Amount exceeds maximum allowed value")

if cents == 0:

    raise ValidationError("Amount must be greater than zero")

return cents

except (InvalidOperation, ValueError) as e:

    raise ValidationError(f"Invalid amount format: {amount}")
```

```
def validate_currency(currency: str) -> str:
```

```
"""
```

```
Validates ISO 4217 currency code.
```

Args:

```
    currency: Three-letter currency code (e.g., "USD")
```

Returns:

```
    str: Uppercase currency code
```

Raises:

```
    ValidationError: If currency code is invalid
```

```
"""
```

```
if not isinstance(currency, str):
    raise ValidationError("Currency must be a string")

currency = currency.upper().strip()

if len(currency) != 3:
    raise ValidationError("Currency code must be exactly 3 characters")

if not currency.isalpha():
    raise ValidationError("Currency code must contain only letters")

if currency not in VALID_CURRENCIES:
    raise ValidationError(f"Unsupported currency: {currency}")

return currency

def validate_idempotency_key(key: str) -> str:
    """
    Validates idempotency key format.

    Keys should be unique, reasonably long, and contain safe characters.
    """
```

Args:

key: Client-provided idempotency key

Returns:

str: Validated idempotency key

Raises:

ValidationError: If key format is invalid

"""

```
if not isinstance(key, str):

    raise ValidationError("Idempotency key must be a string")

key = key.strip()

if len(key) < 16:

    raise ValidationError("Idempotency key must be at least 16 characters")

if len(key) > 255:

    raise ValidationError("Idempotency key cannot exceed 255 characters")

# Allow alphanumeric, hyphens, underscores

if not re.match(r'^[a-zA-Z0-9_-]+$', key):

    raise ValidationError(
        "Idempotency key can only contain letters, numbers, hyphens, and underscores"
    )

return key
```

Complete custom exceptions:

```
# utils/exceptions.py                                         PYTHON

class PaymentGatewayError(Exception):

    """Base exception for all payment gateway errors."""

    pass


class ValidationError(PaymentGatewayError):

    """Raised when input validation fails."""

    pass


class IdempotencyError(PaymentGatewayError):

    """Raised when idempotency key conflicts occur."""

    pass


class PaymentProcessingError(PaymentGatewayError):

    """Raised when payment processing fails."""

    pass


class WebhookVerificationError(PaymentGatewayError):

    """Raised when webhook signature verification fails."""

    pass


class ReconciliationError(PaymentGatewayError):

    """Raised when state reconciliation detects inconsistencies."""

    pass
```

D. Core Logic Skeleton Code (signature + TODOs only):

Here are the key service interfaces that learners will implement:

```
# services/payment_intent_manager.py

from typing import Optional, Dict, Any

from datetime import datetime

from models.payment_intent import PaymentIntent, PaymentIntentStatus

from repositories.payment_intent_repository import PaymentIntentRepository

from utils.validation import validate_amount, validate_currency, validate_idempotency_key


class PaymentIntentManager:

    def __init__(self, repository: PaymentIntentRepository):

        self.repository = repository


    def create_intent(
            self,
            amount: int,
            currency: str,
            idempotency_key: str,
            metadata: Optional[Dict[str, Any]] = None
        ) -> PaymentIntent:

        """
        Creates a new payment intent with idempotency protection.

        Returns existing intent if idempotency key already exists.

        Creates new intent if key is unique.

        Raises IdempotencyError if key exists with different parameters.

        """

        # TODO 1: Validate all input parameters using validation utilities

        # TODO 2: Check if payment intent already exists with this idempotency_key

        # TODO 3: If exists, compare parameters - return existing if same, error if different

        # TODO 4: If not exists, create new PaymentIntent with status 'created'

        # TODO 5: Generate client_secret for secure intent confirmation
```

```

# TODO 6: Save intent to repository within database transaction

# TODO 7: Return created PaymentIntent

pass


def get_intent(self, intent_id: str) -> Optional[PaymentIntent]:
    """Retrieves payment intent by ID."""

    # TODO 1: Query repository for intent by ID

    # TODO 2: Return intent if found, None if not found

    pass


def update_intent_status(
    self,
    intent_id: str,
    new_status: PaymentIntentStatus
) -> PaymentIntent:
    """
    Updates payment intent status with validation.

    Ensures valid state transitions according to state machine.
    """

    # TODO 1: Retrieve current intent from repository

    # TODO 2: Validate state transition is allowed (created -> processing, etc.)

    # TODO 3: Update intent status and updated_at timestamp

    # TODO 4: Save updated intent to repository

    # TODO 5: Return updated intent

    pass


def expire_stale_intents(self, older_than: datetime) -> int:
    """
    Background job to cancel payment intents older than specified time.

    Returns count of expired intents.

```

```
....  
  
# TODO 1: Query repository for intents in 'created' status older than cutoff  
  
# TODO 2: For each stale intent, update status to 'canceled'  
  
# TODO 3: Bulk update repository with canceled intents  
  
# TODO 4: Return count of expired intents  
  
pass
```

```
# services/webhook_processor.py

from typing import Dict, Any, Optional

from models.webhook_event import WebhookEvent

from repositories.webhook_event_repository import WebhookEventRepository

from utils.crypto import verify_webhook_signature

class WebhookProcessor:

    def __init__(self, repository: WebhookEventRepository):
        self.repository = repository
        self.webhook_secret = os.getenv("WEBHOOK_SECRET")

    def process_webhook(
            self,
            payload: str,
            signature: str,
            event_type: str
        ) -> Optional[WebhookEvent]:
        """
        Processes incoming webhook with signature verification and deduplication.

        Returns processed webhook event or None if duplicate/invalid.
        """

        # TODO 1: Verify webhook signature using HMAC-SHA256
        # TODO 2: Parse JSON payload and extract event ID
        # TODO 3: Check if event ID already processed (deduplication)
        # TODO 4: If duplicate, return existing WebhookEvent record
        # TODO 5: If new, create WebhookEvent record with parsed data
        # TODO 6: Route to appropriate handler based on event_type
        # TODO 7: Update payment status based on webhook event
        # TODO 8: Mark webhook as processed and save to repository
```

```
pass

def verify_signature(self, payload: str, signature: str) -> bool:
    """Verifies webhook payload signature using HMAC."""
    # TODO 1: Extract timestamp and signature from header
    # TODO 2: Check timestamp is within acceptable window (5 minutes)
    # TODO 3: Compute HMAC-SHA256 of payload using webhook secret
    # TODO 4: Compare computed signature with provided signature
    # TODO 5: Return True if signatures match, False otherwise
    pass
```

E. Language-Specific Hints:

- **Database Sessions:** Use FastAPI's `Depends(get_db)` for automatic session management. Sessions are automatically closed after request completion.
- **Async vs Sync:** Start with synchronous SQLAlchemy for simplicity. Async can be added later for performance optimization.
- **Environment Variables:** Use `python-dotenv` to load `.env` files during development. Production should use container environment variables.
- **JSON Serialization:** Pydantic schemas automatically handle JSON serialization/deserialization with validation.
- **Error Handling:** FastAPI automatically converts Python exceptions to HTTP error responses when using `HTTPException`.
- **Database Migrations:** Use Alembic (SQLAlchemy's migration tool) for production database schema changes.
- **Testing Database:** Use SQLite in-memory database for fast tests: `sqlite:///memory:`

F. Milestone Checkpoint:

After implementing the high-level architecture foundation:

Validation Checkpoint:

```
python -m pytest tests/test_validation.py -v
```

BASH

Expected output: All validation tests pass, covering amount conversion, currency validation, and idempotency key format checking.

Database Checkpoint:

```
python -c "from config.database import create_tables; create_tables(); print('Database tables created successfully')"
```

BASH

Expected output: No errors, database tables created for PaymentIntent, Charge, Refund, and WebhookEvent.

API Startup Checkpoint:

```
uvicorn main:app --reload
```

BASH

Expected output: FastAPI server starts on <http://localhost:8000> with interactive API docs available at <http://localhost:8000/docs>.

Manual Integration Test: Send a test request to create a payment intent:

```
curl -X POST "http://localhost:8000/payment_intents" \
-H "Content-Type: application/json" \
-d '{
    "amount": 1050,
    "currency": "USD",
    "idempotency_key": "test-key-12345678901234"
}'
```

BASH

Expected response: JSON with payment intent ID, status "created", and client_secret field.

G. Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
Database connection errors	Missing environment variables or wrong DATABASE_URL	Check echo=True in engine config, verify connection string	Update DATABASE_URL, ensure PostgreSQL is running
Validation errors not appearing	FastAPI not using Pydantic schemas	Check endpoint parameter types	Add proper type hints and Pydantic models
SQLAlchemy relationship errors	Missing foreign key constraints or back_populates	Check model definitions and relationships	Add proper ForeignKey fields and relationship() calls
Import errors between modules	Circular imports or incorrect Python path	Use python -c "import module_name" to test	Restructure imports, use dependency injection
Tests failing with database issues	Test database not isolated	Check test fixtures and database setup	Use separate test database or in-memory SQLite

Data Model

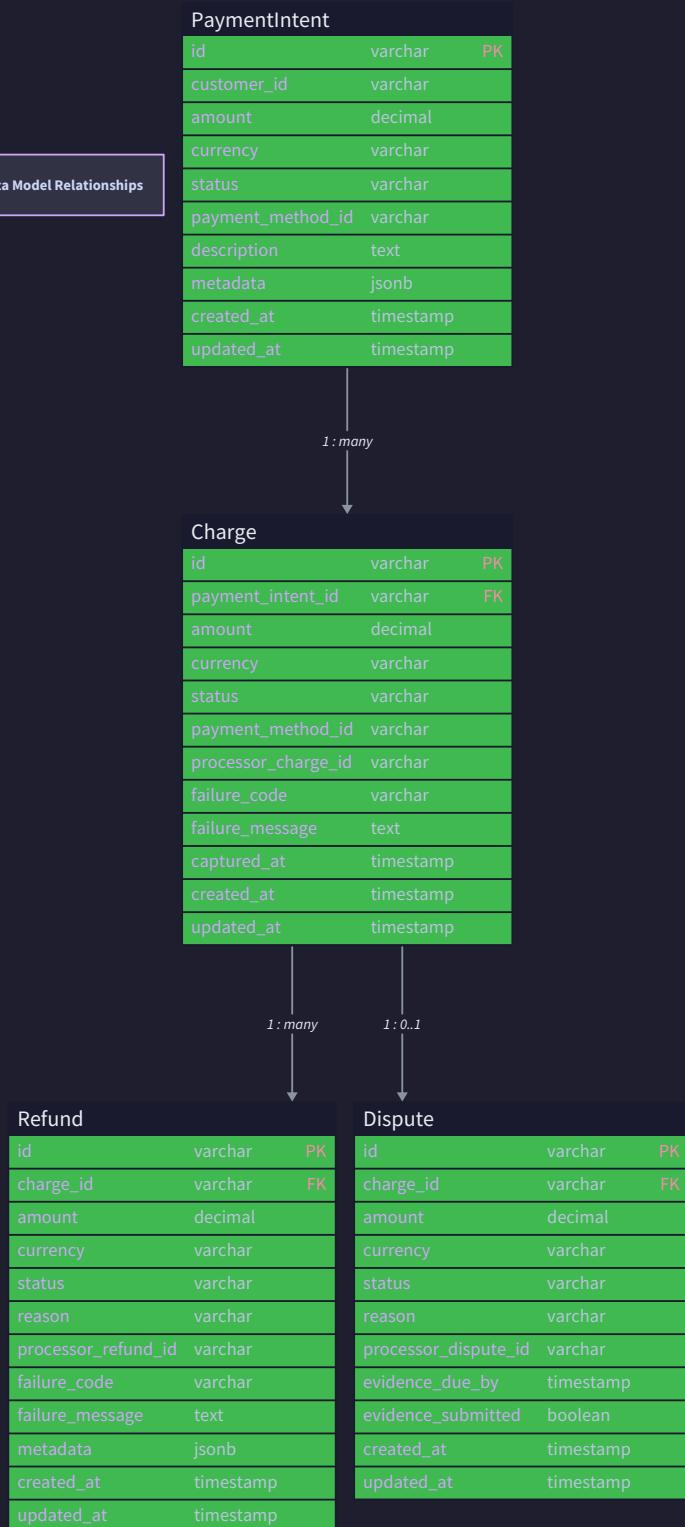
Milestone(s): Foundation for all milestones - data structures support payment intents (Milestone 1), payment processing (Milestone 2), refunds and disputes (Milestone 3), and webhook reconciliation (Milestone 4)

The payment gateway data model serves as the foundation for tracking money movement through complex, multi-step financial transactions. Think of the data model as a comprehensive audit trail for a bank - every action, state change, and

monetary movement must be precisely recorded with immutable timestamps and references. Unlike typical application data that can be updated in-place, payment data follows append-only patterns where historical states are preserved for regulatory compliance and dispute resolution.

The core challenge in payment data modeling is handling the inherent asynchrony of financial systems. A single customer payment intent spawns multiple interconnected records across different tables as the transaction flows through various stages: intent creation, authorization, capture, settlement, and potential refunds or disputes. Each stage may occur minutes, hours, or even days apart, requiring careful relationship management and state consistency.

Payment Data Model Relationships



Payment Data Model

Core Entities:

- **PaymentIntent**: Customer's commitment to pay
- **Charge**: Actual money movement attempt
- **Refund**: Money returned to customer
- **Dispute**: Customer challenges charge

Key Principles:

- Append-only for audit trails
- Immutable transaction history

- Immutable transaction history
- Foreign key relationships maintain data integrity

Core Payment Entities

The payment system's data model centers around four primary entities that represent different aspects of the payment lifecycle. These entities form a hierarchical relationship where each level adds specificity and operational detail to the layer above.

Payment Intent represents the customer's commitment to pay before any money actually moves. Think of a payment intent as a formal IOU that establishes the transaction context, amount, currency, and associated metadata. The intent serves as the immutable anchor point that persists throughout the entire payment lifecycle, providing a stable identifier that remains constant even as the underlying payment methods, amounts, or processing details change.

Field	Type	Description
<code>id</code>	<code>str</code>	Globally unique payment intent identifier (pi_xxxx format)
<code>amount</code>	<code>int</code>	Payment amount in smallest currency unit (cents for USD)
<code>currency</code>	<code>str</code>	ISO 4217 three-letter currency code (USD, EUR, GBP)
<code>status</code>	<code>PaymentIntentStatus</code>	Current state in the payment intent lifecycle
<code>idempotency_key</code>	<code>str</code>	Client-provided unique key for duplicate request prevention
<code>metadata</code>	<code>dict</code>	Arbitrary key-value data attached by merchant application
<code>created_at</code>	<code>datetime</code>	Immutable timestamp when intent was first created
<code>expires_at</code>	<code>datetime</code>	Automatic cancellation timestamp for stale intents
<code>client_secret</code>	<code>str</code>	Secret token for client-side payment confirmation
<code>last_payment_error</code>	<code>dict</code>	Most recent error encountered during payment processing

The `amount` field stores monetary values as integers in the smallest currency unit to eliminate floating-point precision errors. For USD, \$10.00 becomes 1000 cents. The `idempotency_key` enables safe retry behavior - if a client sends the same key twice, they receive the same response rather than creating duplicate payments.

Charge represents the actual movement of money from customer to merchant for a specific payment intent. While a payment intent declares the intention to pay, a charge executes that intention by submitting payment details to the card network or bank. Multiple charges may be associated with a single payment intent if the first charge fails and requires retry with different payment methods.

Field	Type	Description
<code>id</code>	<code>str</code>	Globally unique charge identifier (ch_xxxx format)
<code>payment_intent_id</code>	<code>str</code>	Foreign key reference to associated payment intent
<code>amount</code>	<code>int</code>	Actual charged amount in smallest currency unit
<code>status</code>	<code>ChargeStatus</code>	Current processing status of the charge
<code>payment_method_token</code>	<code>str</code>	Tokenized reference to customer payment method
<code>failure_code</code>	<code>str</code>	Standardized error code if charge failed
<code>failure_message</code>	<code>str</code>	Human-readable error description if charge failed
<code>network_transaction_id</code>	<code>str</code>	Payment network's identifier for this transaction
<code>authorization_code</code>	<code>str</code>	Bank authorization code for successful charges
<code>created_at</code>	<code>datetime</code>	Timestamp when charge was initiated
<code>settled_at</code>	<code>datetime</code>	Timestamp when funds were actually transferred

The `payment_method_token` field contains an opaque reference to the customer's payment details rather than storing sensitive card information directly. This tokenization approach maintains PCI DSS compliance by ensuring sensitive data never touches the merchant's systems.

Refund represents the return of previously charged funds back to the customer. Refunds maintain a direct relationship to their originating charge and track the complex state transitions involved in reversing money movement through banking networks. Partial refunds are supported, allowing merchants to return portions of the original charge while retaining the remainder.

Field	Type	Description
<code>id</code>	<code>str</code>	Globally unique refund identifier (re_xxxx format)
<code>charge_id</code>	<code>str</code>	Foreign key reference to the charge being refunded
<code>amount</code>	<code>int</code>	Refund amount in smallest currency unit
<code>status</code>	<code>RefundStatus</code>	Current processing status of the refund
<code>reason</code>	<code>str</code>	Merchant-provided reason for the refund
<code>network_transaction_id</code>	<code>str</code>	Payment network identifier for refund transaction
<code>failure_code</code>	<code>str</code>	Error code if refund processing failed
<code>failure_message</code>	<code>str</code>	Human-readable error description if refund failed
<code>created_at</code>	<code>datetime</code>	Timestamp when refund was initiated
<code>processed_at</code>	<code>datetime</code>	Timestamp when refund completed processing

Webhook Event captures all payment-related notifications received from external payment providers. These events serve as the authoritative source of truth for payment state changes, providing the external perspective on transaction status that must be reconciled with internal records.

Field	Type	Description
<code>id</code>	<code>str</code>	Globally unique webhook event identifier
<code>event_type</code>	<code>str</code>	Standardized event type (payment_intent.succeeded, charge.failed)
<code>payload</code>	<code>dict</code>	Complete event data received from payment provider
<code>signature</code>	<code>str</code>	HMAC signature for webhook authenticity verification
<code>processed_at</code>	<code>datetime</code>	Timestamp when webhook was successfully processed
<code>processing_attempts</code>	<code>int</code>	Number of times processing was attempted
<code>last_processing_error</code>	<code>str</code>	Most recent error encountered during processing
<code>source_ip</code>	<code>str</code>	IP address that sent the webhook (for security logging)
<code>created_at</code>	<code>datetime</code>	Timestamp when webhook was first received

Design Insight: The webhook event entity serves dual purposes - it provides idempotency for webhook processing (preventing duplicate handling of the same event) and creates an audit trail for troubleshooting payment discrepancies. Every webhook is preserved permanently to support regulatory compliance and dispute resolution.

Dispute represents customer-initiated chargebacks where the cardholder contests a charge through their issuing bank. Disputes introduce significant complexity because they operate outside the normal payment flow and can occur months after the original transaction. The dispute entity tracks the complex lifecycle of chargeback management including evidence submission deadlines and resolution outcomes.

Field	Type	Description
<code>id</code>	<code>str</code>	Globally unique dispute identifier (dp_xxxx format)
<code>charge_id</code>	<code>str</code>	Foreign key reference to the disputed charge
<code>amount</code>	<code>int</code>	Disputed amount in smallest currency unit
<code>status</code>	<code>DisputeStatus</code>	Current stage of dispute processing
<code>reason</code>	<code>str</code>	Cardholder's stated reason for the dispute
<code>evidence_due_by</code>	<code>datetime</code>	Deadline for submitting dispute evidence
<code>evidence_submitted</code>	<code>bool</code>	Whether merchant evidence was submitted
<code>network_reason_code</code>	<code>str</code>	Payment network's standardized dispute code
<code>created_at</code>	<code>datetime</code>	Timestamp when dispute was first reported
<code>resolved_at</code>	<code>datetime</code>	Timestamp when dispute reached final resolution

State Enumerations

Payment processing requires carefully managed state machines to track the progression of financial transactions through their complex lifecycles. Each entity maintains its own state enumeration with strictly defined transition rules that prevent invalid state changes and ensure data consistency.

Decision: Explicit State Enumerations vs. Boolean Flags

- **Context:** Payment entities need to track their progression through multi-step processes with complex state transitions
- **Options Considered:** Boolean flags (is_pending, is_completed), status strings, enumerated states
- **Decision:** Use explicit enumerated states with controlled transitions
- **Rationale:** Enumerations prevent invalid state combinations, make state transitions explicit, and provide type safety for state-dependent operations
- **Consequences:** Requires more upfront design but eliminates entire classes of state consistency bugs

Option	Pros	Cons	Chosen?
Boolean Flags	Simple to implement, flexible combinations	Allows invalid states, no transition control	✗
String Status	Easy to extend, human-readable	No type safety, typo-prone, no validation	✗
Explicit Enums	Type-safe, controlled transitions, clear semantics	More upfront design, migration complexity	✓

Payment Intent Status governs the high-level lifecycle of a customer's commitment to pay. The state machine ensures that payment intents progress logically from creation through completion or cancellation without skipping intermediate states or moving backwards inappropriately.

State	Description	Valid Transitions	Terminal State
created	Intent established but no payment processing attempted	requires_action, processing, canceled	✗
requires_action	Customer action needed (3D Secure authentication)	processing, canceled	✗
processing	Payment being processed by external provider	succeeded, requires_action, canceled	✗
succeeded	Payment completed successfully	None	✓
canceled	Payment intent canceled before completion	None	✓

The `requires_action` state handles Strong Customer Authentication scenarios where additional customer interaction is required. Payment intents in this state wait for customer completion of 3D Secure challenges or similar authentication flows.

Charge Status tracks the execution of money movement for individual payment attempts. Charges have a simpler state machine than payment intents because they represent discrete processing attempts rather than ongoing customer

interactions.

State	Description	Valid Transitions	Terminal State
pending	Charge submitted to payment processor	succeeded, failed	✗
succeeded	Funds successfully captured from customer	None	✓
failed	Charge processing failed permanently	None	✓

Refund Status manages the complex process of returning funds through banking networks. Refund processing involves multiple parties (merchant, payment processor, issuing bank, customer bank) and can fail at various stages, requiring careful state tracking.

State	Description	Valid Transitions	Terminal State
pending	Refund initiated but not yet processed	succeeded, failed	✗
succeeded	Refund completed and funds returned to customer	None	✓
failed	Refund processing failed permanently	None	✓

Dispute Status tracks the extended lifecycle of chargeback management, which can span months and involve multiple rounds of evidence submission and review by card networks.

State	Description	Valid Transitions	Terminal State
warning_needs_response	Pre-dispute warning requiring merchant response	under_review, charge_refunded	✗
under_review	Evidence submitted, awaiting network decision	won, lost	✗
charge_refunded	Merchant refunded to avoid formal dispute	None	✓
won	Merchant successfully defended the dispute	None	✓
lost	Dispute resolved in favor of cardholder	None	✓

Critical Insight: State transitions must be validated at the database level using check constraints to prevent application bugs from creating invalid state combinations. Invalid states in payment processing can lead to regulatory violations and financial losses.

Entity Relationships

The payment data model implements a hierarchical relationship structure where higher-level entities (payment intents) provide context and stability while lower-level entities (charges, refunds) handle operational details and state changes.

These relationships must maintain referential integrity even under high concurrency and system failures.

Payment Intent to Charge Relationship follows a one-to-many pattern where a single payment intent may generate multiple charges if initial attempts fail or require different payment methods. This relationship enables robust retry logic while maintaining a consistent customer experience.

The foreign key relationship uses `payment_intent_id` in the charge table referencing `id` in the payment intent table. Database constraints prevent orphaned charges and ensure every charge has a valid parent intent. The relationship includes cascade rules that handle cleanup when payment intents are archived or deleted.

`PaymentIntent (1) ← (Many) Charge`

- One intent can have multiple charge attempts
- `Charge.payment_intent_id → PaymentIntent.id`
- ON DELETE CASCADE: removing intent removes all associated charges
- Unique constraint on `(payment_intent_id, sequence_number)` orders attempts

Charge to Refund Relationship implements one-to-many semantics supporting partial refunds where multiple refund transactions may apply to a single charge. The relationship includes validation constraints ensuring total refund amounts never exceed the original charge amount.

`Charge (1) ← (Many) Refund`

- One charge can have multiple partial refunds
- `Refund.charge_id → Charge.id`
- ON DELETE RESTRICT: charges with refunds cannot be deleted
- Check constraint: $\text{SUM}(\text{refunds.amount}) \leq \text{charge.amount}$

Charge to Dispute Relationship follows one-to-one semantics since card networks typically allow only one dispute per charge. However, the relationship uses one-to-many modeling to accommodate edge cases where disputes are reopened or multiple dispute types apply to the same transaction.

`Charge (1) ← (Many) Dispute`

- Typically one dispute per charge, but supports multiple for edge cases
- `Dispute.charge_id → Charge.id`
- ON DELETE RESTRICT: disputed charges cannot be deleted
- Check constraint: Only one active dispute per charge

Webhook Event Relationships use polymorphic references where a single webhook event may relate to any of the core payment entities depending on the event type. The relationship uses a combination of event type and entity ID to establish connections.

`WebhookEvent → PaymentIntent/Charge/Refund/Dispute`

- `Event.payload` contains `entity_id` and `entity_type`
- No direct foreign key due to polymorphic nature
- Application-level validation ensures referenced entities exist
- Index on `(entity_type, entity_id, event_type)` for efficient lookups

Decision: Direct Foreign Keys vs. Polymorphic References for Webhooks

- **Context:** Webhook events can reference any type of payment entity based on the event content
- **Options Considered:** Separate webhook tables per entity, polymorphic columns, JSON payload references
- **Decision:** Use JSON payload references with application-level validation
- **Rationale:** Webhook payloads are inherently polymorphic and change frequently; rigid foreign key constraints would require constant schema migrations
- **Consequences:** Enables flexible webhook handling but requires careful application-level referential integrity validation

Referential Integrity Constraints ensure data consistency across the entire payment system. These constraints prevent common data corruption scenarios and provide strong guarantees about relationship validity.

Constraint Type	Implementation	Purpose
Foreign Key	<code>charge.payment_intent_id → payment_intent.id</code>	Prevent orphaned charges
Foreign Key	<code>refund.charge_id → charge.id</code>	Prevent orphaned refunds
Foreign Key	<code>dispute.charge_id → charge.id</code>	Prevent orphaned disputes
Check	<code>SUM(refund.amount WHERE charge_id = X) ≤ charge.amount</code>	Prevent over-refunding
Check	<code>payment_intent.amount > 0</code>	Ensure positive payment amounts
Check	<code>refund.amount > 0</code>	Ensure positive refund amounts
Unique	<code>(payment_intent.idempotency_key)</code>	Prevent duplicate intents
Unique	<code>(webhook_event.id, event_type)</code>	Prevent duplicate webhook processing

Cascade Behavior defines what happens to related entities when parent records are modified or deleted. Payment systems require careful cascade design to balance data integrity with regulatory preservation requirements.

Parent Entity	Child Entity	Delete Behavior	Update Behavior
PaymentIntent	Charge	CASCADE	CASCADE
Charge	Refund	RESTRICT	CASCADE
Charge	Dispute	RESTRICT	CASCADE
PaymentIntent	WebhookEvent	NO ACTION	NO ACTION

The RESTRICT behavior on charges with refunds or disputes prevents accidental deletion of financially significant records. Webhook events use NO ACTION because they represent historical audit data that must be preserved regardless of entity lifecycle changes.

Common Pitfalls in Entity Relationships:

⚠ Pitfall: Allowing Null Foreign Keys in Financial Records Financial entities should never have orphaned records due to null foreign keys. Every charge must belong to a payment intent, every refund must reference a charge. Null foreign keys in payment processing indicate data corruption and can lead to reconciliation failures. Use NOT NULL constraints on all relationship columns and validate relationships before insertion.

⚠ Pitfall: Insufficient Cascade Rules for Payment Intent Cleanup Payment intents that fail early in the lifecycle may accumulate without proper cleanup, leading to database bloat and performance degradation. Implement time-based cascade rules that automatically remove failed payment intents after appropriate retention periods, ensuring associated charges and events are cleaned up appropriately.

⚠ Pitfall: Missing Referential Integrity for Polymorphic Webhook References Since webhooks use JSON payload references rather than foreign keys, application code must validate that referenced entities exist before processing events. Implement validation triggers or application-level checks to ensure webhook events never reference non-existent payment entities, preventing webhook processing failures.

Implementation Guidance

The payment data model requires careful implementation to handle the unique requirements of financial data: immutability, auditability, and strict consistency guarantees. The recommended approach uses a combination of relational database features and application-level validation to ensure data integrity.

Technology Recommendations:

Component	Simple Option	Advanced Option
Database	PostgreSQL with JSON columns	PostgreSQL with separate audit tables
ORM	SQLAlchemy with declarative models	SQLAlchemy with event listeners for audit trails
Migrations	Alembic with manual schema changes	Alembic with automated rollback testing
Validation	Pydantic models with custom validators	Pydantic with database constraint validation
Testing	In-memory SQLite for unit tests	Docker PostgreSQL for integration tests

Recommended File Structure:

```
payment_gateway/
├── models/
│   ├── __init__.py
│   ├── base.py          ← Base model with common fields
│   ├── payment_intent.py ← PaymentIntent model and status enum
│   ├── charge.py        ← Charge model and status enum
│   ├── refund.py         ← Refund model and status enum
│   ├── dispute.py       ← Dispute model and status enum
│   └── webhook_event.py ← WebhookEvent model
├── database/
│   ├── __init__.py
│   ├── session.py        ← Database session management
│   └── constraints.py    ← Custom constraint validators
├── validators/
│   ├── __init__.py
│   ├── amount.py          ← Amount validation functions
│   └── currency.py        ← Currency validation functions
└── migrations/
    └── alembic/           ← Database migration files
```

Base Model Infrastructure (Complete Implementation):

```
# models/base.py

from datetime import datetime

from typing import Optional

import uuid

from sqlalchemy import Column, String, DateTime, func

from sqlalchemy.ext.declarative import declarative_base

from sqlalchemy.dialects.postgresql import UUID

Base = declarative_base()

class BaseModel(Base):

    __abstract__ = True

    id = Column(String, primary_key=True, default=lambda: str(uuid.uuid4()))

    created_at = Column(DateTime, nullable=False, default=func.now())

    updated_at = Column(DateTime, nullable=False, default=func.now(), onupdate=func.now())

    def __repr__(self):

        return f"<{self.__class__.__name__}(id='{self.id}')>"

# database/session.py

from contextlib import contextmanager

from sqlalchemy import create_engine

from sqlalchemy.orm import sessionmaker, Session

from typing import Generator

# Database configuration

DATABASE_URL = "postgresql://user:password@localhost/payment_gateway"

engine = create_engine(DATABASE_URL, echo=False, pool_pre_ping=True)

SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

@contextmanager
```

```
def get_db_session() -> Generator[Session, None, None]:  
    """Database session dependency with automatic cleanup."""  
  
    session = SessionLocal()  
  
    try:  
  
        yield session  
  
        session.commit()  
  
    except Exception:  
  
        session.rollback()  
  
        raise  
  
    finally:  
  
        session.close()  
  
  
def get_db() -> Session:  
    """FastAPI dependency for database sessions."""  
  
    return SessionLocal()
```

Validation Functions (Complete Implementation):

```
# validators/amount.py

def validate_amount(amount: int) -> int:
    """
    Validates payment amount ensuring positive value in smallest currency unit.
    """
```

Args:

amount: Amount in cents (or smallest currency unit)

Returns:

Validated amount as integer

Raises:

ValueError: If amount is negative or zero

"""

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

if amount <= 0:
    raise ValueError("Amount must be positive")

if amount > 99999999: # $999,999.99 limit
    raise ValueError("Amount exceeds maximum allowed value")

return amount
```

```
# validators/currency.py
```

```
SUPPORTED_CURRENCIES = {"USD", "EUR", "GBP", "CAD", "AUD", "JPY"}
```

```
def validate_currency(currency: str) -> str:
    """
```

Validates ISO 4217 currency code.

Args:

currency: Three-letter currency code

Returns:

Validated currency code in uppercase

Raises:

ValueError: If currency is invalid or unsupported

"""

```
if not isinstance(currency, str):
```

```
    raise ValueError("Currency must be a string")
```

```
currency = currency.upper().strip()
```

```
if len(currency) != 3:
```

```
    raise ValueError("Currency must be 3-letter ISO 4217 code")
```

```
if currency not in SUPPORTED_CURRENCIES:
```

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

```
return currency
```

```
def validate_idempotency_key(key: str) -> str:
```

"""

Validates idempotency key format and length.

Args:

key: Client-provided idempotency key

Returns:

Validated idempotency key

```
Raises:
```

```
    ValueError: If key is invalid format or too long/short

"""

if not isinstance(key, str):
    raise ValueError("Idempotency key must be a string")

key = key.strip()

if len(key) < 1:
    raise ValueError("Idempotency key cannot be empty")

if len(key) > 255:
    raise ValueError("Idempotency key too long (max 255 characters)")

# Allow alphanumeric, hyphens, underscores
if not key.replace('-', '').replace('_', '').isalnum():
    raise ValueError("Idempotency key contains invalid characters")

return key
```

Core Model Skeletons (Signatures + TODOs):

```
# models/payment_intent.py

from enum import Enum

from sqlalchemy import Column, String, Integer, DateTime, JSON

from sqlalchemy.orm import relationship

from .base import BaseModel


class PaymentIntentStatus(Enum):

    CREATED = "created"

    REQUIRES_ACTION = "requires_action"

    PROCESSING = "processing"

    SUCCEEDED = "succeeded"

    CANCELED = "canceled"

class PaymentIntent(BaseModel):

    __tablename__ = "payment_intents"

    # TODO 1: Add amount column as Integer, not null

    # TODO 2: Add currency column as String(3), not null

    # TODO 3: Add status column as Enum(PaymentIntentStatus), not null, default=CREATED

    # TODO 4: Add idempotency_key column as String(255), not null, unique

    # TODO 5: Add metadata column as JSON, nullable=True

    # TODO 6: Add client_secret column as String, not null

    # TODO 7: Add expires_at column as DateTime, nullable=True

    # TODO 8: Add last_payment_error column as JSON, nullable=True

    # Relationships

    # TODO 9: Add charges relationship to Charge model with back_populates

    # TODO 10: Add cascade="all, delete-orphan" for charge cleanup

    def can_transition_to(self, new_status: PaymentIntentStatus) -> bool:

        """Check if status transition is valid according to state machine."""


```

```

# TODO 11: Implement state transition validation

# TODO 12: Use dictionary mapping current_status -> [allowed_next_statuses]

# TODO 13: Return True if transition allowed, False otherwise

pass


def is_terminal_status(self) -> bool:
    """Check if current status is terminal (no further transitions)."""

    # TODO 14: Return True for succeeded/canceled, False for others

    pass


# models/charge.py

class ChargeStatus(Enum):

    PENDING = "pending"

    SUCCEEDED = "succeeded"

    FAILED = "failed"


class Charge(BaseModel):

    __tablename__ = "charges"

    # TODO 1: Add payment_intent_id as String, ForeignKey, not null

    # TODO 2: Add amount column as Integer, not null

    # TODO 3: Add status column as Enum(ChargeStatus), not null, default=PENDING

    # TODO 4: Add payment_method_token column as String, not null

    # TODO 5: Add failure_code column as String, nullable=True

    # TODO 6: Add failure_message column as String, nullable=True

    # TODO 7: Add network_transaction_id column as String, nullable=True

    # TODO 8: Add authorization_code column as String, nullable=True

    # TODO 9: Add settled_at column as DateTime, nullable=True

    # Relationships

    # TODO 10: Add payment_intent relationship with back_populates

```

```
# TODO 11: Add refunds relationship to Refund model  
  
# TODO 12: Add disputes relationship to Dispute model
```

Database Constraint Implementation:

```
# database/constraints.py  
  
from sqlalchemy import event, func  
  
from sqlalchemy.exc import IntegrityError  
  
@event.listens_for(Refund, 'before_insert')  
  
@event.listens_for(Refund, 'before_update')  
  
def validate_refund_amount(mapper, connection, target):  
  
    """Ensure refund amount doesn't exceed remaining charge balance."""  
  
    # TODO 1: Query total existing refunds for target.charge_id  
  
    # TODO 2: Calculate remaining refundable amount  
  
    # TODO 3: Raise IntegrityError if new refund exceeds remaining  
  
    # TODO 4: Handle case where charge doesn't exist  
  
    pass  
  
@event.listens_for(PaymentIntent, 'before_update')  
  
def validate_status_transition(mapper, connection, target):  
  
    """Ensure payment intent status transitions follow state machine rules."""  
  
    # TODO 5: Get current status from database  
  
    # TODO 6: Check if transition to target.status is valid  
  
    # TODO 7: Raise IntegrityError for invalid transitions  
  
    # TODO 8: Allow same-status "transitions" (no-op updates)  
  
    pass
```

Milestone Checkpoints:

After implementing the core data models, validate the implementation:

1. **Database Creation Test:** Run `alembic upgrade head` to create all tables with constraints
2. **Model Validation Test:** Create test instances of each model with valid and invalid data
3. **Relationship Test:** Create payment intent with charges, verify foreign key relationships work
4. **Constraint Test:** Attempt to create invalid refund amounts, verify constraints prevent it

5. State Transition Test:

Try invalid status transitions, ensure they're blocked

Expected behavior: All valid operations succeed, invalid operations raise appropriate exceptions with clear error messages. Database constraints should prevent data corruption even if application validation is bypassed.

Debugging Tips:

Symptom	Likely Cause	Diagnosis	Fix
Foreign key constraint errors	Missing parent record or invalid ID format	Check if referenced payment_intent/charge exists	Validate parent exists before creating child
Refund amount validation fails	Total refunds exceed original charge	Query <code>SUM(refunds.amount)</code> for the charge	Implement remaining balance check
Duplicate idempotency key errors	Client retrying with same key but different data	Check if existing record matches new request exactly	Return existing record for true duplicates, error for conflicts
Status transition errors	Invalid state machine transition attempted	Log current and target status	Implement proper state transition validation

Payment Intent Management

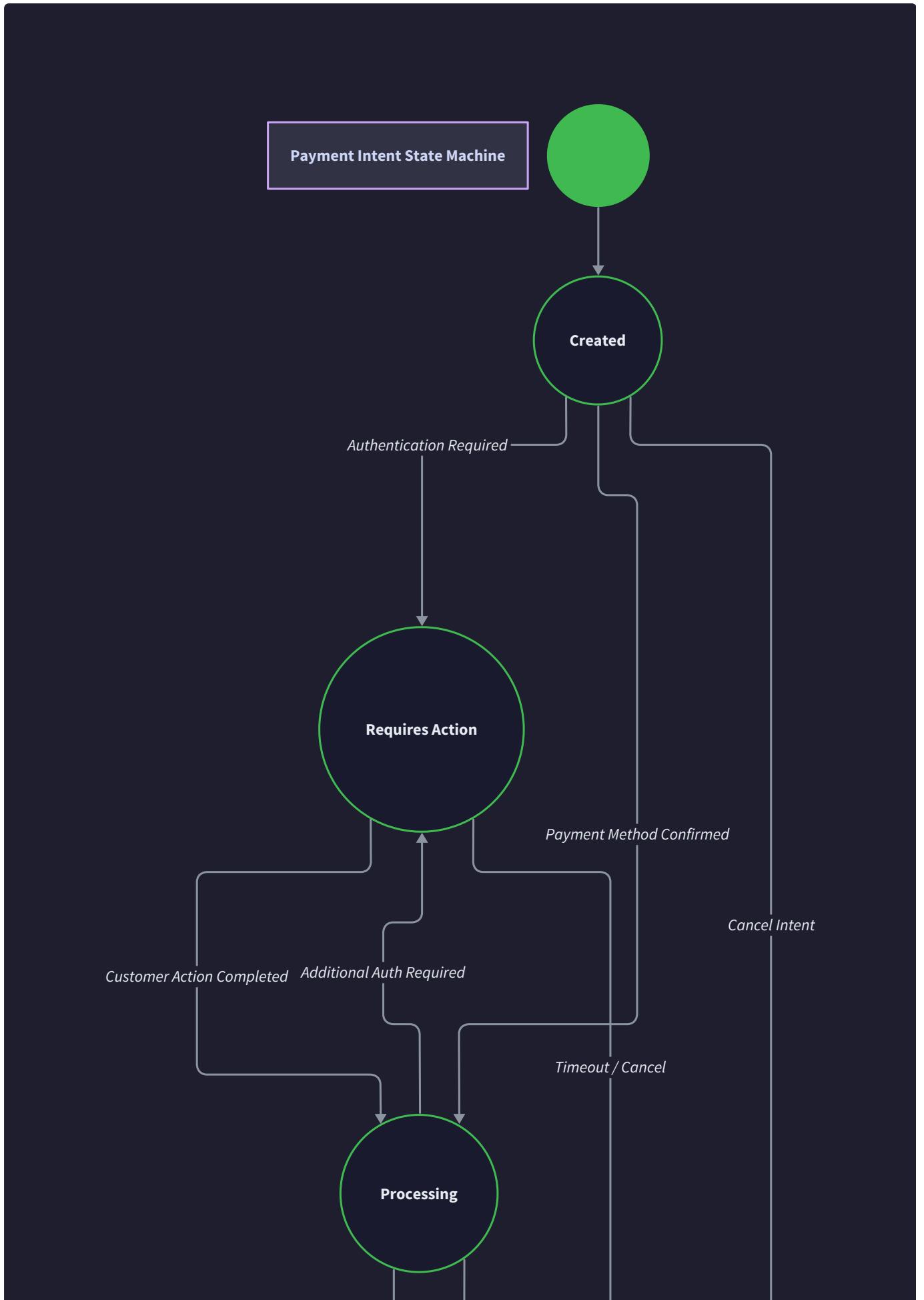
Milestone(s): Milestone 1: Payment Intent & Idempotency - implementing payment intents with idempotency keys to prevent duplicate charges, tracking payment state transitions, and handling request deduplication

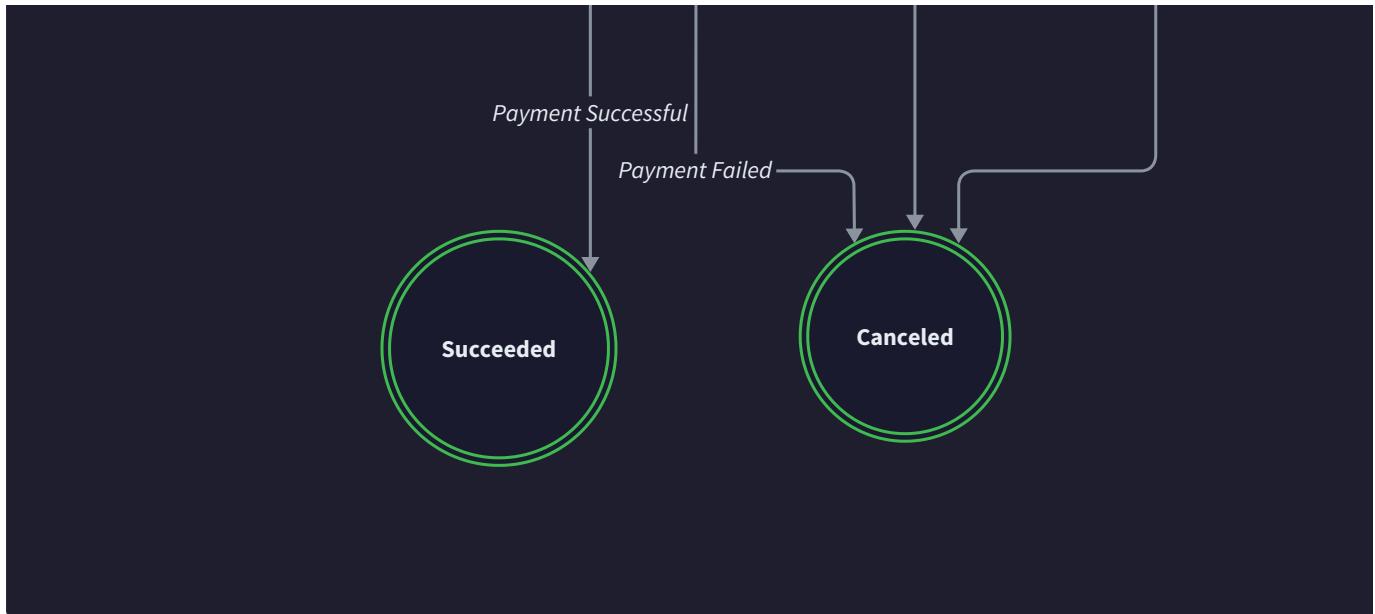
Think of a **payment intent** as a customer's firm commitment to buy something before any money actually moves - like signing a contract to purchase a house before the bank transfer happens. Just as that purchase contract has a lifecycle (signed, inspected, financed, closed), payment intents move through defined states from creation to completion. The critical insight is that creating the intent is separate from processing the payment, which allows us to handle complex scenarios like authentication challenges, network failures, and duplicate requests without accidentally charging customers multiple times.

Payment intent management forms the foundation of reliable payment processing by establishing a clear contract between merchant and customer before attempting to move money. This separation of intent from execution enables sophisticated error handling, prevents duplicate charges through idempotency controls, and provides a clear audit trail for every payment attempt.

Payment Intent Lifecycle

The payment intent state machine governs how payment requests progress from initial creation through final resolution. Think of this like a restaurant order workflow - the order starts as "placed" (customer committed), moves to "preparing" (kitchen working), might require "customer confirmation" (special requests), and finally reaches "completed" or "cancelled". Each transition is controlled and reversible only within certain windows.





State Definitions and Transitions:

Current State	Valid Next States	Trigger Event	Business Logic	Rollback Allowed
<code>created</code>	<code>requires_action</code> , <code>processing</code> , <code>canceled</code>	Client confirms payment, 3DS required, timeout/cancellation	Initial state after intent creation	Yes - no money movement yet
<code>requires_action</code>	<code>processing</code> , <code>canceled</code>	Customer completes 3DS, timeout/cancellation	Waiting for customer authentication	Yes - authentication not completed
<code>processing</code>	<code>succeeded</code> , <code>canceled</code>	Provider confirms/rejects charge	Payment submitted to provider	Limited - may be too late
<code>succeeded</code>	None (terminal)	Provider settles successfully	Money transferred, receipt generated	No - only via separate refund
<code>canceled</code>	None (terminal)	Explicit cancellation or timeout	Intent abandoned or expired	No - final state

The state machine enforces critical business rules that prevent money movement until all preconditions are met. For example, an intent cannot jump directly from `created` to `succeeded` - it must pass through `processing` to ensure proper provider interaction and audit logging.

The Critical Insight: Terminal states (`succeeded` , `canceled`) are permanent by design. Once money has moved or an intent has been definitively abandoned, the intent becomes immutable. This prevents accidental state corruption and provides clear audit boundaries.

State Transition Validation Logic:

Each state transition must satisfy specific preconditions to maintain system integrity. The validation rules ensure that payment intents follow realistic business workflows and prevent impossible state combinations.

Validation Rule	Check Performed	Failure Action
Monotonic progression	New state must be "later" in lifecycle	Reject transition with error
Terminal state immutability	Cannot modify succeeded or canceled intents	Return current state unchanged
Required field presence	Intent must have amount, currency, payment method	Block transition until complete
Expiration boundary	Cannot process expired intents	Force transition to canceled
Concurrent modification	Only one state change per intent at a time	Use database row locking

Intent Metadata and Context:

Payment intents carry extensive metadata that enriches the payment context and supports advanced features like fraud detection, customer service, and analytics. This metadata remains mutable throughout the intent lifecycle, allowing merchants to add context as the payment progresses.

Metadata Category	Fields	Purpose	Mutability
Customer Context	customer_id, email, billing_address	Identity and fraud checks	Read-only after creation
Order Details	order_id, line_items, shipping_address	Business context and fulfillment	Updateable until processing
Payment Context	payment_method_types, setup_future_usage	Processing preferences	Read-only after creation
Merchant Data	statement_descriptor, receipt_email	Customer-facing information	Updateable until succeeded
System Tracking	application_fee, transfer_data, on_behalf_of	Marketplace and platform data	Read-only after creation

Decision: State Machine Implementation Pattern

- **Context:** Need to enforce valid state transitions while supporting concurrent access and preventing race conditions
- **Options Considered:**
 - Enum-based validation with application logic
 - Database constraints with state transition table
 - Event sourcing with state reconstruction
- **Decision:** Database constraints with application-level validation wrapper
- **Rationale:** Combines database-level integrity guarantees with application flexibility, supports concurrent access through row locking, and provides clear audit trail
- **Consequences:** Enables safe concurrent processing, requires careful transaction management, adds complexity to deployment (schema migrations)

Idempotency Key Design

Idempotency keys solve the fundamental challenge of network unreliability in payment processing. Think of them like check numbers on paper checks - each check has a unique number, and if you accidentally submit the same check twice to your bank, the second submission is rejected rather than processed again. Similarly, idempotency keys ensure that network retries, browser refreshes, or accidental double-clicks don't result in multiple charges.

The idempotency system creates a binding between a client-provided unique identifier and a specific payment operation. Once that binding is established, any future request with the same idempotency key returns the result of the original operation rather than creating a new payment.

Idempotency Key Lifecycle and Storage:

Phase	Key Status	Database State	Client Behavior	System Response
First Request	New	Key + request hash stored	Provides key + payment params	Process normally, store result
Duplicate Request (same params)	Exists, params match	Key + request hash + result stored	Same key + same params	Return stored result
Conflicting Request	Exists, params differ	Key + different request hash found	Same key + different params	Return 400 error - key reused incorrectly
Expired Key	Exists but expired	Key marked expired in database	Same key after expiration	Allow new request with same key
Processing Request	Exists, no result yet	Key stored, payment in progress	Same key while processing	Return current payment status

The core challenge is distinguishing between legitimate retries (same operation, network hiccup) and incorrect key reuse (same key, different operation). We solve this by hashing the request parameters and storing that hash alongside the idempotency key.

Request Parameter Hashing Strategy:

To detect parameter conflicts, we hash the essential payment parameters that define the operation's intent. The hash must be stable (same inputs always produce same hash) but sensitive to meaningful changes.

Parameter Category	Included in Hash	Excluded from Hash	Rationale
Core Payment Data	<code>amount</code> , <code>currency</code> , <code>customer_id</code>	<code>metadata</code> , <code>description</code>	Financial impact vs. contextual info
Payment Method	<code>payment_method_id</code> , <code>payment_method_types</code>	<code>receipt_email</code> , <code>statement_descriptor</code>	Payment execution vs. presentation
System Context	<code>application_fee_amount</code> , <code>transfer_data</code>	<code>expand</code> , <code>trace_id</code>	Business logic vs. debugging/display
Timing Data	None	<code>created_at</code> , <code>last_updated</code>	Parameters shouldn't include server-generated timestamps

Key Insight: The parameter hash serves as a "fingerprint" of the payment intent. If two requests have the same idempotency key but different fingerprints, they represent different business operations and the second should be rejected to prevent accidental misuse.

Idempotency Key Generation and Validation:

Clients must provide idempotency keys that are truly unique and sufficiently random to prevent accidental collisions. The system validates key format and provides guidance for proper generation.

Validation Rule	Requirement	Example Valid	Example Invalid	Error Message
Length	22-255 characters	<code>idem_1234567890abcdef</code>	<code>abc</code> (too short)	"Idempotency key must be 22-255 characters"
Character Set	Alphanumeric, underscore, dash	<code>order-123_retry-1</code>	<code>key with spaces!</code>	"Idempotency key contains invalid characters"
Uniqueness	No active duplicates in system	First use of any string	Reused key with different params	"Idempotency key already used with different parameters"
Client Prefix	Recommended client identifier	<code>mobile_app_ prefix</code>	<code>Generic key123</code>	Warning: "Consider adding client prefix for easier debugging"

Database Schema for Idempotency Tracking:

The idempotency table acts as a distributed lock and result cache, ensuring exactly-once semantics across concurrent requests.

Field	Type	Constraints	Purpose
idempotency_key	varchar(255)	Primary key, not null	Client-provided unique identifier
request_hash	varchar(64)	Not null, index	SHA-256 of essential request parameters
payment_intent_id	varchar(255)	Foreign key, nullable initially	Links to created payment intent
response_body	jsonb	Nullable	Cached response for completed requests
response_status	integer	Nullable	HTTP status of cached response
created_at	timestamp	Not null, default now()	When idempotency key first used
completed_at	timestamp	Nullable	When response was cached
expires_at	timestamp	Not null, index	When key can be reused
client_ip	inet	Nullable	Request source for debugging

Decision: Idempotency Window Duration

- **Context:** Need to balance preventing duplicate charges with allowing legitimate key reuse after reasonable time
- **Options Considered:**
 - 24 hours (Stripe's approach)
 - 1 hour (aggressive cleanup)
 - 7 days (conservative approach)
- **Decision:** 24 hours for most operations, 1 hour for failed operations
- **Rationale:** 24 hours handles delayed retries and batch processing, 1 hour for failures allows quick recovery from transient issues without blocking legitimate retries
- **Consequences:** Requires background cleanup process, balances safety with usability, follows industry standard

Intent Expiration Handling

Payment intents have natural lifespans - customers don't hold shopping carts forever, and payment methods can become invalid over time. Think of intent expiration like restaurant reservations - if you don't show up within a reasonable window, the table gets released to other customers. Intent expiration prevents resource leaks, cleans up abandoned sessions, and ensures payment methods remain valid when processing occurs.

The expiration system operates on multiple timescales: short-term expiration for active user sessions, medium-term expiration for abandoned intents, and long-term cleanup for historical records. Each expiration tier serves different business and technical requirements.

Expiration Timing and Business Rules:

Intent State	Default Expiration	Business Justification	System Behavior
<code>created</code>	60 minutes	User actively in checkout flow	Transition to <code>canceled</code> , notify client
<code>requires_action</code>	30 minutes	Waiting for 3DS authentication	Transition to <code>canceled</code> , release payment method
<code>processing</code>	10 minutes	Provider should respond quickly	Investigate with provider, potential retry
<code>succeeded</code>	Never expires	Permanent record for accounting	Archive to cold storage after 7 years
<code>canceled</code>	Archive after 90 days	Fraud analysis and debugging	Move to historical archive

The expiration times balance user experience (don't expire too quickly) with resource management (don't hold resources indefinitely). Different states have different urgency levels based on system resource consumption and user expectations.

Background Expiration Processing:

The expiration system runs as a background service that periodically scans for expired intents and performs appropriate cleanup actions. This service must handle high volumes efficiently while maintaining data consistency.

Processing Stage	Operation	Batch Size	Frequency	Error Handling
Scan for Expired	Query intents where <code>expires_at < now()</code>	1000 records	Every 5 minutes	Log failures, continue with rest of batch
State Transition	Update expired intents to <code>canceled</code>	100 per transaction	As needed	Rollback transaction on failure
Cleanup Resources	Release payment method tokens, clear cache	10 concurrent	As needed	Retry individual failures
Archive Records	Move old canceled intents to archive table	500 records	Daily at 2 AM	Alert operations team on failure
Update Metrics	Track expiration rates by state and reason	All processed records	Real-time	Non-blocking, best-effort

Expiration Event Notification:

When payment intents expire, various system components need notification to clean up related resources and update external systems.

Notification Target	Event Type	Payload	Delivery Method	Retry Policy
Client Application	payment_intent.canceled	Intent ID, expiration reason	Webhook	3 retries with exponential backoff
Analytics System	intent_expired	Intent data, duration, state	Event stream	At-least-once delivery
Customer Service	High-value intent expired	Customer info, order details	Alert queue	Immediate, with escalation
Fraud Detection	Suspicious expiration pattern	IP, timing, behavior	Real-time API	Synchronous, fast timeout
Cache Layer	invalidate_intent	Intent ID	Redis pub/sub	Fire-and-forget

Graceful Expiration vs. Hard Timeout:

The system implements a two-stage expiration process: graceful expiration that allows completion of in-flight operations, followed by hard timeout that forces cleanup regardless of state.

Expiration Type	Trigger Condition	Grace Period	Actions Allowed	Force Cleanup
Graceful	expires_at reached	5 minutes	Complete current operation, no new operations	No - allow natural completion
Hard Timeout	Grace period exceeded	None	No operations allowed	Yes - force state transition
Emergency	System overload or security threat	None	Immediate termination	Yes - aggressive cleanup

During graceful expiration, the intent can still transition to succeeded if a payment is actively processing. This prevents race conditions where a successful charge is incorrectly canceled due to timing.

Intent Extension and Renewal:

Some business scenarios require extending payment intent lifespans beyond default expiration times. The system supports controlled extension with proper authorization and limits.

Extension Reason	Max Extensions	Extension Duration	Authorization Required	Audit Trail
User Request	2 extensions	30 minutes each	Customer authentication	Full request/response log
System Retry	5 extensions	10 minutes each	Automated based on error type	Error details and retry count
Customer Service	No limit	Up to 24 hours	Support agent approval	Agent ID and business justification
Fraud Investigation	No limit	Up to 7 days	Security team approval	Investigation case number

Design Principle: Expiration is a safety mechanism, not a user-hostile feature. The goal is preventing resource leaks and stale state while maximizing successful payment completion rates.

Common Pitfalls in Intent Expiration:

⚠ **Pitfall: Race Condition Between Expiration and Success** When an intent expires while a payment is processing, the system might cancel the intent just as the payment succeeds, leading to a successful charge with a canceled intent. Fix this by checking payment status before forcing expiration and allowing a brief grace period for in-flight transactions.

⚠ **Pitfall: Cascading Expiration Failures** If the expiration background service fails, expired intents accumulate and can overwhelm the system when it recovers. Implement circuit breakers and gradual ramp-up when processing large backlogs of expired intents.

⚠ **Pitfall: Timezone Confusion in Expiration Logic** Storing expiration times in local timezone or mixing timezone-aware and naive timestamps leads to incorrect expiration behavior. Always use UTC for internal expiration tracking and convert to local time only for display purposes.

⚠ **Pitfall: Insufficient Expiration Monitoring** High expiration rates might indicate UX problems, system performance issues, or integration bugs, but without proper monitoring, these problems go undetected. Track expiration rates by state, duration, and cause to identify systematic issues.

Implementation Guidance

Technology Recommendations:

Component	Simple Option	Advanced Option	Rationale
Database	PostgreSQL with JSONB	PostgreSQL with separate tables for metadata	JSONB provides flexibility while maintaining ACID properties
Caching	Redis for idempotency keys	Redis Cluster for high availability	Fast lookups for duplicate detection
Background Jobs	Celery with Redis broker	Celery with RabbitMQ	Reliable task processing for expiration cleanup
Monitoring	Python logging + Prometheus	Structured logging + DataDog	Essential for payment system observability
API Framework	FastAPI with Pydantic	FastAPI with async PostgreSQL driver	Type safety and excellent OpenAPI documentation

File Structure:

```
payment_gateway/
├── models/
│   ├── __init__.py
│   ├── payment_intent.py      ← PaymentIntent model and state machine
│   └── idempotency.py        ← IdempotencyKey model and validation
├── services/
│   ├── __init__.py
│   ├── payment_intent_service.py ← Core business logic
│   └── idempotency_service.py  ← Duplicate detection logic
├── api/
│   ├── __init__.py
│   └── payment_intents.py    ← REST API endpoints
├── workers/
│   ├── __init__.py
│   └── expiration_worker.py  ← Background expiration processing
├── database/
│   ├── __init__.py
│   └── migrations/           ← Database schema changes
└── tests/
    ├── __init__.py
    ├── test_payment_intent_service.py
    └── test_idempotency_service.py
```

Infrastructure Starter Code:

Complete database models with all required fields and relationships:

```
# models/payment_intent.py

from datetime import datetime, timedelta

from enum import Enum

from typing import Dict, Any, Optional

from sqlalchemy import Column, String, Integer, DateTime, JSON, Text

from sqlalchemy.ext.declarative import declarative_base

from sqlalchemy.dialects.postgresql import UUID

import uuid

Base = declarative_base()

class PaymentIntentStatus(str, Enum):
    CREATED = "created"
    REQUIRES_ACTION = "requires_action"
    PROCESSING = "processing"
    SUCCEEDED = "succeeded"
    CANCELED = "canceled"

class PaymentIntent(Base):
    __tablename__ = "payment_intents"

    id = Column(String, primary_key=True, default=lambda: f"pi_{uuid.uuid4().hex}")
    amount = Column(Integer, nullable=False) # Amount in cents
    currency = Column(String(3), nullable=False) # ISO 4217 currency code
    status = Column(String, nullable=False, default=PaymentIntentStatus.CREATED.value)
    idempotency_key = Column(String(255), nullable=False, unique=True)
    metadata = Column(JSON, nullable=False, default=dict)
    created_at = Column(DateTime, nullable=False, default=datetime.utcnow)
    expires_at = Column(DateTime, nullable=False,
                        default=lambda: datetime.utcnow() + timedelta(hours=1))
    client_secret = Column(String, nullable=False,
```

```

        default=lambda: f"pi_{uuid.uuid4().hex}_secret_{uuid.uuid4().hex}")

last_payment_error = Column(JSON, nullable=True)

def can_transition_to(self, new_status: PaymentIntentStatus) -> bool:
    """Validates if transition to new status is allowed"""

    # TODO: Implement state transition validation logic

    # Valid transitions defined in state machine diagram

    pass

def is_terminal_status(self) -> bool:
    """Checks if current status allows no further transitions"""

    return self.status in [PaymentIntentStatus.SUCCEEDED.value,
                           PaymentIntentStatus.CANCELED.value]

# models/idempotency.py

class IdempotencyKey(Base):
    __tablename__ = "idempotency_keys"

    idempotency_key = Column(String(255), primary_key=True)
    request_hash = Column(String(64), nullable=False)
    payment_intent_id = Column(String, nullable=True)
    response_body = Column(JSON, nullable=True)
    response_status = Column(Integer, nullable=True)
    created_at = Column(DateTime, nullable=False, default=datetime.utcnow)
    completed_at = Column(DateTime, nullable=True)
    expires_at = Column(DateTime, nullable=False,
                        default=lambda: datetime.utcnow() + timedelta(hours=24))
    client_ip = Column(String, nullable=True)

```

Complete validation utilities for payment parameters:

```
# services/validation.py

import hashlib

import json

import re

from typing import Dict, Any

from decimal import Decimal, ROUND_HALF_UP

SUPPORTED_CURRENCIES = {"USD", "EUR", "GBP", "CAD", "AUD", "JPY"}


def validate_amount(amount: Any) -> int:

    """Converts monetary amount to cents with validation"""

    # TODO 1: Handle string inputs by converting to Decimal for precision

    # TODO 2: Validate amount is positive and not zero

    # TODO 3: Convert to cents (multiply by 100) and round properly

    # TODO 4: Ensure result fits in 32-bit signed integer

    # TODO 5: Raise ValueError with descriptive message for invalid amounts

    pass


def validate_currency(currency: str) -> str:

    """Validates ISO 4217 currency code"""

    # TODO 1: Check currency is string and exactly 3 characters

    # TODO 2: Convert to uppercase for consistency

    # TODO 3: Verify currency code exists in SUPPORTED_CURRENCIES

    # TODO 4: Raise ValueError if currency not supported

    pass


def validate_idempotency_key(key: str) -> str:

    """Validates idempotency key format and requirements"""

    # TODO 1: Check key is string and within length limits (22-255 chars)

    # TODO 2: Validate character set (alphanumeric, underscore, dash only)

    # TODO 3: Ensure key has sufficient entropy (not obviously sequential)

    # TODO 4: Return normalized key (trimmed whitespace)
```

```
pass

def hash_request_parameters(params: Dict[str, Any]) -> str:

    """Creates stable hash of essential request parameters"""

    # TODO 1: Extract only parameters that affect payment processing

    # TODO 2: Sort parameters for consistent hash regardless of order

    # TODO 3: Convert to canonical JSON representation

    # TODO 4: Generate SHA-256 hash and return as hex string

    essential_params = {

        "amount": params.get("amount"),

        "currency": params.get("currency"),

        "customer_id": params.get("customer_id"),

        "payment_method_id": params.get("payment_method_id"),

        "application_fee_amount": params.get("application_fee_amount"),

    }

    # Filter out None values and create stable hash

    pass
```

Core Logic Skeleton:

Payment intent service with complete method signatures and detailed TODOs:

```
# services/payment_intent_service.py

from typing import Dict, Any, Optional, Tuple

from sqlalchemy.orm import Session

from models.payment_intent import PaymentIntent, PaymentIntentStatus

from models.idempotency import IdempotencyKey

from services.validation import validate_amount, validate_currency, validate_idempotency_key


class PaymentIntentService:

    def __init__(self, db_session: Session):
        self.db = db_session

    def create_payment_intent(self,
                             amount: Any,
                             currency: str,
                             idempotency_key: str,
                             metadata: Dict[str, Any] = None) -> Tuple[PaymentIntent, bool]:
        """
        Creates new payment intent with idempotency protection.

        Returns (payment_intent, was_created) tuple.

        """
        # TODO 1: Validate all input parameters using validation functions
        # TODO 2: Check if idempotency key already exists in database
        # TODO 3: If key exists, validate request parameters match stored hash
        # TODO 4: If parameters don't match, raise IdempotencyError with details
        # TODO 5: If key exists with matching params, return existing intent
        # TODO 6: If key is new, create new PaymentIntent with validated data
        # TODO 7: Store IdempotencyKey record with request hash
        # TODO 8: Commit transaction and return (intent, True) for new intent
        # Hint: Use database transaction to ensure atomicity
        pass
```

```
def transition_intent_status(self,
                                intent_id: str,
                                new_status: PaymentIntentStatus,
                                error_details: Dict[str, Any] = None) -> PaymentIntent:
    """
    Safely transitions payment intent to new status with validation.

    """
    # TODO 1: Load payment intent with row-level locking (SELECT FOR UPDATE)
    # TODO 2: Check intent exists and is not in terminal state
    # TODO 3: Validate transition is allowed using can_transition_to()
    # TODO 4: Update intent status and last_payment_error if provided
    # TODO 5: Save changes and commit transaction
    # TODO 6: Return updated intent
    # Hint: Use pessimistic locking to prevent concurrent modifications
    pass
```

```
def expire_stale_intents(self, batch_size: int = 100) -> int:
    """
    Background task to expire payment intents past their expiration time.

    Returns count of intents that were expired.

    """
    # TODO 1: Query for intents where expires_at < now() and status not terminal
    # TODO 2: Limit to batch_size to prevent overwhelming database
    # TODO 3: For each intent, transition to CANCELED status
    # TODO 4: Log expiration with intent ID and original status
    # TODO 5: Count successful transitions and return total
    # TODO 6: Handle individual failures gracefully - continue with batch
    # Hint: Process in smaller transactions to avoid long-running locks
    pass
```

```

class IdempotencyService:

    def __init__(self, db_session: Session):
        self.db = db_session


    def check_idempotency_key(self,
                             key: str,
                             request_params: Dict[str, Any]) -> Optional[IdempotencyKey]:
        """
        Checks if idempotency key exists and validates request parameters.

        Returns existing key record or None if key is new.

        """
        # TODO 1: Hash the request parameters for comparison
        # TODO 2: Query database for existing idempotency key
        # TODO 3: If key doesn't exist, return None (new request)
        # TODO 4: If key exists, compare stored request_hash with computed hash
        # TODO 5: If hashes match, return existing key record
        # TODO 6: If hashes don't match, raise IdempotencyError
        # TODO 7: Check if key has expired and handle accordingly
        pass

    class IdempotencyError(Exception):
        """Raised when idempotency key is reused with different parameters"""
        pass

```

Milestone Checkpoint:

After implementing this section, verify the following behavior:

- Payment Intent Creation:** Run `pytest tests/test_payment_intent_service.py::test_create_intent` - should create intent with all required fields populated
- Idempotency Protection:** Make identical API requests twice - second request should return same intent ID and not create duplicate
- Parameter Validation:** Send requests with invalid amounts/currencies - should receive 400 errors with descriptive messages

4. **State Transitions:** Attempt invalid transitions (e.g., `succeeded` to `processing`) - should be rejected
5. **Expiration Processing:** Create intent with 1-second expiration, wait, run expiration worker - intent should transition to `canceled`

Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
Duplicate payments created	Idempotency key validation failing	Check database for multiple intents with same key	Ensure request parameter hashing includes all essential fields
Intent stuck in <code>created</code> status	State transition logic not working	Check database locks and transaction commits	Verify <code>can_transition_to()</code> allows <code>created</code> → <code>processing</code>
Expiration worker not running	Background task configuration	Check Celery worker logs and task queue	Ensure worker is consuming from correct queue
Idempotency errors for same request	Request parameter hash changing	Log computed hashes for identical requests	Exclude non-deterministic fields like timestamps from hash

Payment Processing Engine

Milestone(s): Milestone 2: Payment Processing & 3DS - implementing payment confirmation with 3D Secure authentication flow, processing card payments through external providers, handling authentication redirects, and supporting multiple payment methods

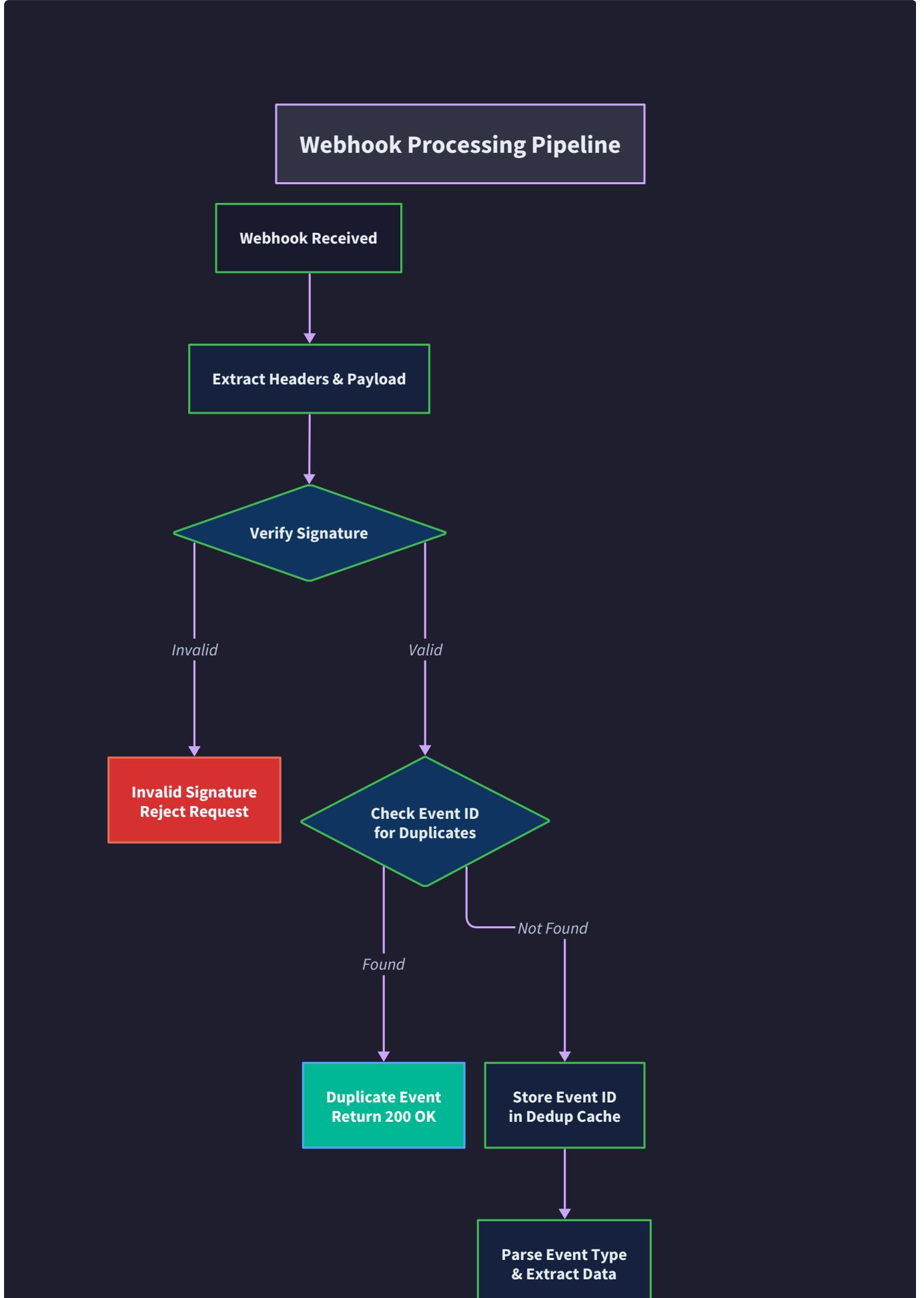
The payment processing engine transforms payment intents into actual money movement by coordinating with external payment providers. Think of this component as the execution engine of a stock trading system - while payment intents represent trading orders waiting to be executed, the processing engine actually submits those orders to the market (payment networks) and handles the complex choreography of authentication, authorization, and settlement.

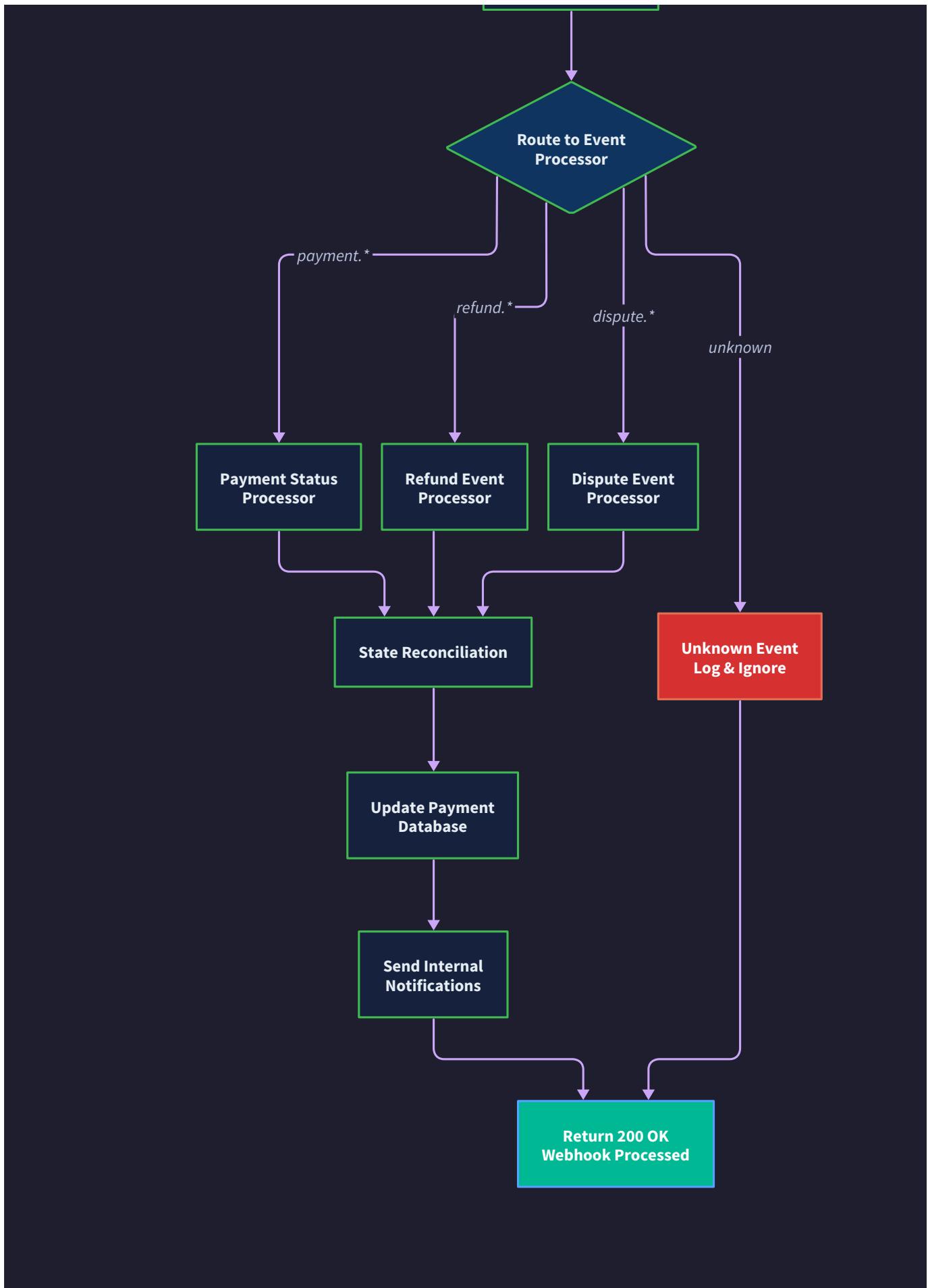
This engine must orchestrate multiple asynchronous operations: tokenizing sensitive payment data, submitting charges to external providers, handling Strong Customer Authentication (3D Secure) redirects, and managing the various success and failure scenarios that can occur during payment processing. Unlike simple API calls, payment processing involves state machines, external redirects, and reconciliation across multiple systems.

The core challenge lies in maintaining payment state consistency while handling the inherently asynchronous nature of modern payment flows. When a customer initiates a payment, the system may need to redirect them to their bank for authentication, wait for external webhook confirmations, and handle various failure modes - all while ensuring the customer has a coherent experience and the merchant's accounting remains accurate.

Charge Processing Flow

The charge processing flow represents the core algorithm that transforms a payment intent into an actual charge against a customer's payment method. Think of this process like conducting an orchestra - multiple instruments (tokenization, provider APIs, authentication flows) must play their parts in precise coordination to create a harmonious payment experience.





The flow begins when a payment intent transitions from the `created` status to active processing. At this point, the system

has already validated the payment amount, currency, and idempotency constraints through the payment intent management layer. The processing engine's responsibility is to execute the actual charge while handling the various authentication and authorization steps required by modern payment regulations.

Decision: Asynchronous Processing with State Persistence

- Context:** Payment processing involves multiple external API calls, potential redirects for authentication, and variable response times from payment providers
- Options Considered:** Synchronous blocking processing, asynchronous with callbacks, event-driven state machine
- Decision:** Asynchronous processing with persistent state tracking through database records
- Rationale:** Payment flows can take minutes (3DS authentication) or hours (bank transfers) to complete, making synchronous processing unsuitable for web applications. State persistence ensures resilience against server restarts and provides audit trails for compliance.
- Consequences:** Enables scalable payment processing and better user experience through non-blocking operations, but requires careful state management and introduces complexity in error handling and recovery scenarios.

The charge processing algorithm follows a predictable sequence of operations designed to maximize success rates while providing clear error reporting for failed transactions:

Processing Step	Purpose	Success Criteria	Failure Handling
Payment Method Validation	Verify tokenized payment method is valid and not expired	Token exists in vault, not marked as expired	Return validation error to client
Provider Selection	Choose appropriate payment provider based on currency, amount, risk factors	Provider supports currency and has sufficient capacity	Fallback to secondary provider or return error
Charge Creation	Submit payment request to external provider with tokenized payment data	Provider returns charge ID and initial status	Log provider error, update charge record with failure details
Authentication Check	Determine if Strong Customer Authentication (3DS) is required	Provider indicates no authentication needed or returns auth challenge	Handle authentication flow or return auth-required status
Authorization Processing	Process the actual payment authorization against customer's account	Provider confirms funds are available and reserved	Update charge status to failed with provider error details
Confirmation Handling	Finalize the charge based on provider response and any authentication results	Charge moves to succeeded status with authorization code	Implement retry logic for transient failures

The processing engine maintains detailed charge records throughout this flow, creating an immutable audit trail of each processing attempt. This audit trail proves essential for debugging payment failures, handling disputes, and maintaining compliance with financial regulations that require transaction traceability.

Charge State Management

Each charge progresses through a well-defined state machine that reflects its current processing status. The `ChargeStatus` enumeration captures three fundamental states that external systems can reliably act upon:

Charge Status	Description	Terminal State	Next Possible States	Typical Duration
<code>pending</code>	Charge submitted to provider, awaiting response	No	succeeded, failed	1-30 seconds for cards, minutes for bank transfers
<code>succeeded</code>	Payment successfully authorized and will settle	Yes	None (refunds create separate records)	N/A - final state
<code>failed</code>	Payment authorization declined or encountered error	Yes	None (retry creates new charge record)	N/A - final state

The charge record captures comprehensive metadata about the processing attempt, enabling detailed analysis of success rates, failure patterns, and provider performance:

Field Name	Type	Description	Usage
<code>id</code>	str	Unique identifier for this specific charge attempt	Primary key, used in refund references
<code>payment_intent_id</code>	str	Foreign key to the originating payment intent	Links charge to customer intent and idempotency tracking
<code>amount</code>	int	Charge amount in smallest currency unit (cents)	Must match or be less than payment intent amount
<code>status</code>	ChargeStatus	Current processing state of the charge	Drives client polling and webhook processing logic
<code>payment_method_token</code>	str	Tokenized reference to customer payment method	Used for provider API calls, never contains raw card data
<code>failure_code</code>	str	Machine-readable error code from provider	Enables automated retry logic and error categorization
<code>failure_message</code>	str	Human-readable error description	Displayed to customers and merchant support teams
<code>network_transaction_id</code>	str	Provider's unique identifier for this transaction	Required for disputes, refunds, and provider support cases
<code>authorization_code</code>	str	Bank authorization code for successful transactions	Proves payment was authorized by issuing bank
<code>created_at</code>	datetime	When charge processing began	Used for timeout detection and performance metrics
<code>settled_at</code>	datetime	When funds were actually transferred	Important for cash flow tracking and reconciliation

Provider Integration Architecture

The processing engine abstracts external payment providers through a standardized interface, enabling support for multiple providers without affecting core payment logic. This abstraction proves critical for merchant resilience, cost optimization, and regulatory compliance across different markets.

The key architectural insight is treating payment providers as interchangeable backends while maintaining consistent internal APIs. This approach mirrors how cloud applications abstract different storage providers - the application logic remains constant while the underlying implementation varies based on operational requirements.

Provider Interface Method	Parameters	Returns	Purpose
<code>create_charge</code>	<code>payment_method_token</code> , <code>amount</code> , <code>currency</code> , <code>metadata</code>	<code>charge_id</code> , <code>status</code> , <code>auth_url</code>	Initiates payment processing with provider
<code>retrieve_charge</code>	<code>provider_charge_id</code>	<code>charge_status</code> , <code>failure_details</code> , <code>settlement_info</code>	Queries current charge state for reconciliation
<code>confirm_payment</code>	<code>provider_charge_id</code> , <code>confirmation_token</code>	<code>final_status</code> , <code>authorization_code</code>	Completes 3DS authentication flow
<code>cancel_charge</code>	<code>provider_charge_id</code> , <code>cancellation_reason</code>	<code>cancellation_status</code> , <code>refunded_amount</code>	Cancels pending charges before settlement

Each provider implementation handles the specific API requirements, authentication schemes, and response formats for their platform while presenting a consistent interface to the processing engine. This design enables A/B testing different providers, implementing failover strategies, and negotiating better rates through competitive provider relationships.

Error Classification and Recovery

Payment processing failures fall into distinct categories that require different handling strategies. The processing engine classifies errors to enable appropriate retry logic, customer communication, and operational alerting:

Error Category	Examples	Retry Strategy	Customer Impact	Operational Response
Transient Provider Errors	Network timeouts, 503 responses, rate limits	Exponential backoff up to 3 attempts	Payment appears as "processing"	Monitor error rates, contact provider if sustained
Permanent Validation Errors	Invalid card number, expired card, insufficient funds	No retry - immediate failure	Clear error message displayed	Update card validation logic if patterns emerge
Authentication Required	3DS challenge needed, strong authentication mandate	Redirect to authentication flow	Redirect to bank authentication page	Monitor authentication success rates
Provider Configuration Issues	Invalid API credentials, unsupported currency	No retry - requires manual intervention	Generic error message	Immediate operational alert, provider configuration review
Fraud Detection Triggers	Unusual spending pattern, high-risk merchant category	Provider-dependent - may require manual review	Payment held for review message	Review fraud rules, potentially whitelist customer

The processing engine implements circuit breaker patterns for provider integrations, temporarily failing fast when provider error rates exceed acceptable thresholds. This prevents cascade failures and provides better user experience during provider outages:

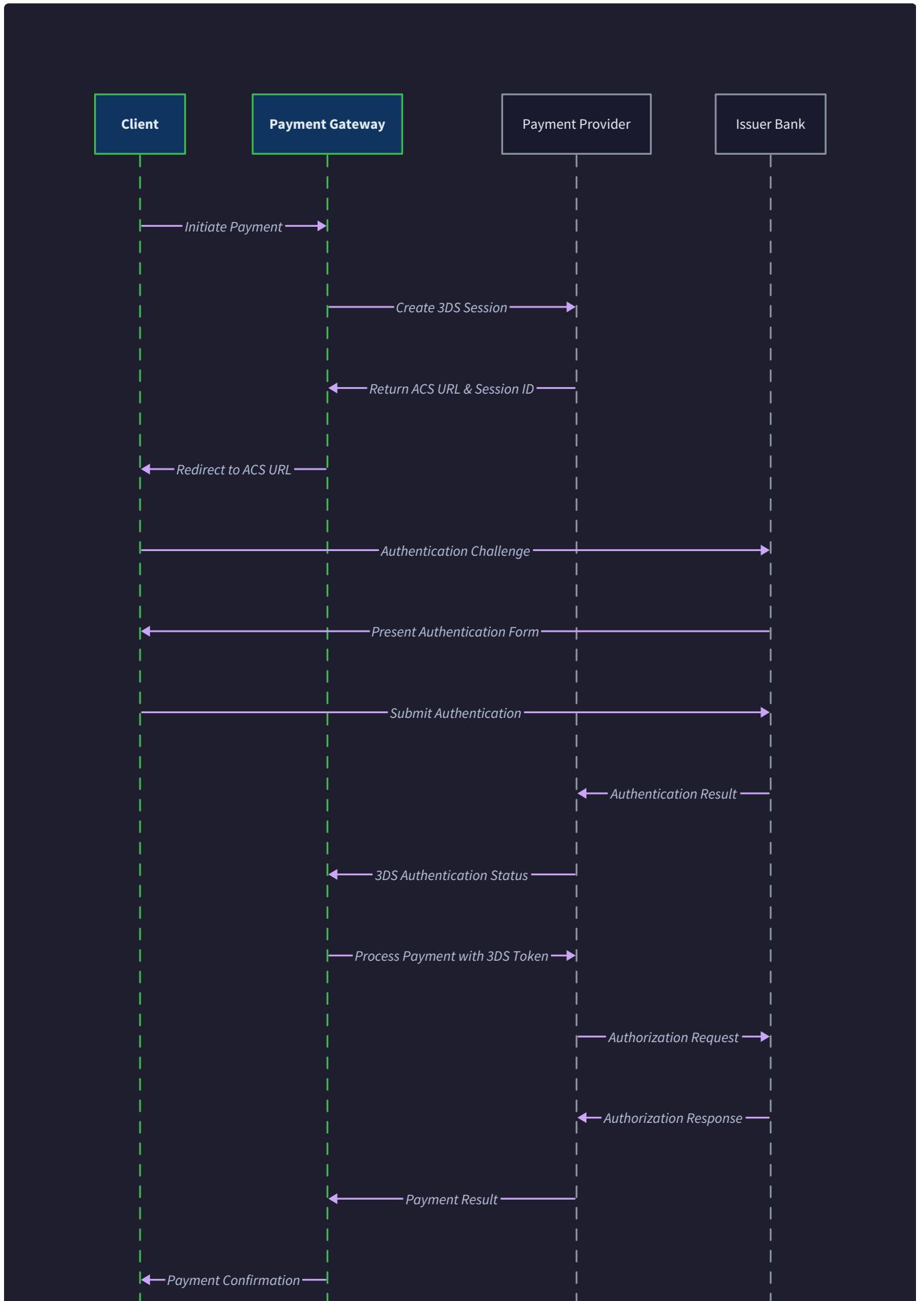
Circuit Breaker States:

1. Closed (normal operation): All requests sent to provider
2. Open (failure mode): Requests immediately failed, provider not called
3. Half-Open (testing): Limited requests sent to test provider recovery

3D Secure Authentication

Strong Customer Authentication through 3D Secure represents one of the most complex aspects of modern payment processing. Think of 3DS as a security checkpoint at an airport - while it adds friction to the payment process, it provides essential fraud protection and regulatory compliance that benefits all parties in the long term.

The 3D Secure protocol enables card issuers to authenticate their cardholders directly during online transactions, reducing fraud liability for merchants while complying with regulations like PSD2 in Europe. However, implementing 3DS requires careful orchestration of redirects, state management, and fallback scenarios to maintain an acceptable user experience.





The authentication flow involves four primary parties: the customer, merchant (our payment gateway), payment provider, and issuing bank. Each party has specific responsibilities and timing requirements that must be carefully coordinated to prevent authentication failures or security vulnerabilities.

3DS Flow Orchestration

The 3D Secure authentication process follows a well-established sequence that balances security requirements with user experience considerations. The processing engine must handle this flow while maintaining payment state consistency and providing clear status updates to the merchant application:

1. **Authentication Assessment:** When creating a charge, the payment provider analyzes transaction risk factors (amount, merchant category, customer history, issuer requirements) to determine if 3DS authentication is mandatory, recommended, or unnecessary.
2. **Challenge Preparation:** If authentication is required, the provider generates a unique authentication session and returns a secure authentication URL along with session parameters that identify this specific payment attempt.
3. **Customer Redirect:** The merchant application redirects the customer to the authentication URL, typically in a popup window or iframe to maintain context of the original purchase flow.
4. **Issuer Authentication:** The customer's bank presents their authentication interface (SMS codes, mobile app push notifications, biometric verification) and validates the customer's identity according to their internal policies.
5. **Authentication Response:** Upon completing authentication, the issuer redirects the customer back to a merchant-provided return URL with authentication results embedded as URL parameters.
6. **Payment Confirmation:** The merchant backend extracts authentication results and calls the provider's confirmation endpoint to complete the original payment using the authentication proof.
7. **Final Settlement:** With authentication confirmed, the provider processes the payment authorization and returns final charge status to complete the payment flow.

The processing engine tracks this multi-step flow through careful state management and timeout handling. Each step has specific failure modes and recovery procedures that must be implemented to provide reliable payment processing.

3DS State Management

During 3D Secure authentication, payment intents transition through specialized states that reflect the current stage of the authentication process. This state tracking enables proper customer communication and prevents duplicate authentication attempts:

Payment Intent Status	3DS Context	Customer Experience	System Behavior
created	Initial state before processing	Payment form submitted, waiting for response	System preparing to submit charge to provider
requires_action	3DS authentication challenge issued	Redirected to bank authentication page	Waiting for customer to complete authentication
processing	Authentication completed, finalizing payment	"Please wait" message while payment completes	Confirming payment with authentication proof
succeeded	Payment authorized and authenticated	Success page displayed	Payment complete, funds will settle
canceled	Authentication failed or timed out	Error message with retry option	Payment attempt recorded as failed

The `requires_action` status proves particularly important for merchant integrations, as it indicates the payment flow requires customer interaction rather than backend processing. Merchant applications must handle this status by presenting the authentication interface rather than polling for completion.

Authentication Challenge Handling

Modern 3D Secure implementations support multiple authentication methods depending on the customer's bank and device capabilities. The processing engine must handle these variations while providing consistent merchant APIs:

Authentication Method	User Experience	Technical Implementation	Fallback Strategy
Frictionless	No customer interaction, authentication happens behind scenes	Provider returns immediate success/failure	None needed - standard processing continues
SMS Challenge	Customer enters code sent to registered phone number	Redirect to issuer page with SMS input form	Allow multiple SMS requests with rate limiting
Mobile App Push	Customer approves transaction in banking mobile app	Redirect shows "Check your mobile app" message	Timeout after 5 minutes, offer SMS alternative
Browser Redirect	Full redirect to issuer website for complex authentication	Customer leaves merchant site temporarily	Return URL must restore shopping context
Biometric	Fingerprint or face recognition through mobile browser APIs	WebAuthn API integration with device capabilities	Fallback to traditional authentication methods

The processing engine stores authentication context in the charge record to enable proper handling of return flows and timeout scenarios. This context includes the authentication method attempted, timestamps for timeout calculation, and return URL parameters for session restoration.

Return URL Management

3D Secure authentication requires merchants to provide return URLs where customers are redirected after completing authentication. The processing engine must generate unique, secure return URLs that restore payment context and

prevent authentication replay attacks:

Return URL Component	Purpose	Security Consideration	Example Value
Base Domain	Merchant's verified domain for customer return	Must match merchant registration to prevent phishing	<code>https://shop.example.com</code>
Return Endpoint	Specific handler for 3DS authentication results	Should be dedicated endpoint, not general payment handler	<code>/payments/3ds-return</code>
Session Token	Unique identifier linking return to original payment	Must be cryptographically secure and single-use	<code>3ds_sess_1a2b3c4d5e6f</code>
Payment Intent ID	Identifies which payment this authentication completes	Allows state restoration and duplicate detection	<code>pi_1234567890abcdef</code>
Timestamp	When authentication session was created	Enables timeout enforcement and replay prevention	<code>1703123456</code>
Signature	HMAC proving URL authenticity and preventing tampering	Computed from all other parameters using secret key	<code>hmac_sha256_abc123def456</code>

The return URL handler must validate all components, check authentication session expiry, and extract authentication results from provider-supplied parameters. Failed validations should be treated as potential security attacks and logged for investigation.

Critical Security Insight: Return URLs represent a potential attack vector where malicious actors might attempt to bypass authentication by crafting fake return requests. Always validate HMAC signatures and check authentication session state before completing payments.

Payment Method Tokenization

Payment method tokenization transforms sensitive card details into secure, opaque tokens that can be safely stored and transmitted without PCI DSS compliance burden. Think of tokenization like a coat check system at a restaurant - customers surrender their valuable items (card details) in exchange for a numbered ticket (token) that can later be used to retrieve the original item when needed.

The tokenization process removes raw payment card data from our systems while maintaining the ability to process payments, refunds, and recurring charges. This approach dramatically reduces security scope, compliance costs, and the potential impact of data breaches while enabling full payment functionality.

Decision: Provider-Managed Tokenization vs Self-Managed Vault

- **Context:** Payment card data requires strict security controls and PCI DSS compliance that significantly increases operational complexity and cost
- **Options Considered:** Self-managed encryption vault, third-party tokenization service, provider-managed tokenization
- **Decision:** Provider-managed tokenization using payment processor's secure vault services
- **Rationale:** Eliminates PCI DSS scope for card storage, reduces security infrastructure requirements, leverages provider's specialized security expertise and compliance certifications
- **Consequences:** Enables rapid development with reduced compliance burden and security risk, but creates dependency on provider tokenization APIs and limits ability to switch providers easily

Tokenization Architecture

The payment processing engine integrates tokenization at multiple points in the payment flow, ensuring sensitive data never persists in application databases or appears in application logs. This integration requires careful API design to maintain security boundaries while providing intuitive developer experiences.

Tokenization Stage	Input Data	Security Boundary	Output Token	Usage Context
Card Collection	Raw card number, expiry, CVV from customer	Client-side form submission over TLS	Single-use collection token	Temporary token for immediate payment processing
Payment Method Storage	Collection token + customer billing details	Server-to-provider API call	Reusable payment method token	Stored for future payments and subscription billing
Charge Processing	Payment method token + charge details	Internal API with token reference	Provider charge ID	Links token to specific charge attempt
Refund Processing	Original charge ID and refund amount	Provider refund API using charge reference	Refund transaction ID	Enables partial refunds without re-exposing card data

The tokenization architecture maintains a clear separation between temporary collection tokens (valid for minutes) and persistent payment method tokens (valid for years). This separation enables different security policies and lifecycle management for each token type.

Secure Token Collection

Modern tokenization implementations collect card details directly from customer browsers to payment provider systems, ensuring sensitive data never reaches merchant servers. This approach requires careful integration of client-side JavaScript libraries with backend payment processing logic:

The collection process begins when customers enter card details in the merchant's payment form. Instead of submitting this data to the merchant server, client-side JavaScript intercepts the form submission and securely transmits card details directly to the payment provider's tokenization endpoint over encrypted connections.

Collection Step	Client-Side Action	Server-Side Response	Security Measure
Form Initialization	Load provider JavaScript library with merchant API keys	Verify merchant domain against provider whitelist	API keys scoped to domain, no secret credentials in browser
Card Data Entry	Customer types card details in merchant-styled form fields	No server involvement - data stays in browser	Form fields replaced with provider-controlled secure inputs
Tokenization Request	JavaScript sends card data directly to provider tokenization API	Provider validates card format and returns collection token	Card data transmitted over TLS to provider, not merchant
Token Submission	Form submits collection token (not card data) to merchant server	Merchant server receives token for payment processing	Merchant never sees raw card data, only secure token reference

This architecture ensures that card details flow directly from customer browsers to payment provider systems without ever existing in merchant application memory, databases, or log files. The merchant application only handles opaque tokens that cannot be used to reconstruct original card data.

Token Storage and Lifecycle

Payment method tokens require careful lifecycle management to balance security, usability, and regulatory compliance. The processing engine implements token storage patterns that support various merchant use cases while maintaining strict security controls:

Storage Pattern	Use Case	Token Validity	Security Controls	Compliance Impact
Session Tokens	Single checkout process	30 minutes	Automatic expiry, single-use validation	No PCI scope - tokens expire quickly
Customer Tokens	Saved payment methods for registered users	1-2 years or until card expiry	Customer authentication required for token usage	Minimal PCI scope - no cardholder data stored
Subscription Tokens	Recurring billing for ongoing services	Until customer cancellation or card expiry	Additional fraud monitoring, velocity limits	Enhanced monitoring required for unattended payments
Merchant Tokens	Customer service operations and refunds	Tied to original transaction lifecycle	Role-based access controls, audit logging	Administrative access controls and compliance reporting

The token storage system maintains metadata about each tokenized payment method without storing sensitive card details. This metadata enables risk analysis, duplicate detection, and customer experience optimization while preserving security boundaries.

Token Metadata Field	Purpose	Example Value	Security Classification
<code>token_id</code>	Unique identifier for this specific token	<code>pm_1a2b3c4d5e6f</code>	Non-sensitive identifier
<code>customer_id</code>	Links token to specific customer account	<code>cust_789xyz</code>	Customer reference (PII considerations)
<code>last_four_digits</code>	Display purposes and duplicate detection	<code>4242</code>	Non-sensitive for display purposes
<code>card_brand</code>	Payment processing and fraud analysis	<code>visa</code>	Non-sensitive routing information
<code>expiry_month</code>	Token validity and renewal prompting	<code>12</code>	Semi-sensitive - used for token lifecycle
<code>expiry_year</code>	Token validity and renewal prompting	<code>2025</code>	Semi-sensitive - used for token lifecycle
<code>fingerprint</code>	Duplicate payment method detection	<code>fp_abc123def456</code>	Non-sensitive hash for deduplication
<code>created_at</code>	Audit trail and lifecycle management	<code>2023-12-01T10:00:00Z</code>	Non-sensitive timestamp
<code>last_used_at</code>	Security monitoring and cleanup	<code>2023-12-15T14:30:00Z</code>	Non-sensitive usage tracking

Token Security Controls

Payment method tokens, while safer than raw card data, still require robust security controls to prevent unauthorized usage and maintain customer trust. The processing engine implements multiple layers of token protection:

Authentication controls ensure only authorized parties can use payment method tokens. Customer tokens require customer authentication, while merchant tokens require staff authentication with appropriate role-based permissions. These controls prevent token theft from resulting in unauthorized charges.

Authorization controls validate each token usage against business rules and risk factors. Unusual spending patterns, geographic inconsistencies, or velocity violations trigger additional verification steps before processing payments. These controls provide defense-in-depth protection against token abuse.

Security Control	Implementation	Protection Against	Performance Impact
Customer Authentication	JWT tokens or session validation	Unauthorized customer token usage	Minimal - cached authentication state
Role-Based Access	Permission matrix for staff operations	Internal fraud and unauthorized refunds	Low - permission check per request
Fraud Scoring	Machine learning risk analysis	Stolen token usage and synthetic fraud	Medium - ML inference per transaction
Velocity Limits	Rate limiting based on token usage patterns	Automated attacks and bulk fraud	Low - counter increment per usage
Geographic Controls	IP geolocation vs billing address comparison	Cross-border fraud patterns	Low - geolocation lookup per request
Device Fingerprinting	Browser/device characteristics tracking	Account takeover and device spoofing	Medium - fingerprint generation and comparison

The token security system logs all usage attempts, successful authentications, and security violations to provide comprehensive audit trails for compliance reporting and fraud investigation.

Common Pitfalls

⚠ Pitfall: Storing Collection Tokens Long-Term Many implementations mistakenly treat collection tokens as permanent payment method references and attempt to store them for future use. Collection tokens are designed for immediate usage and typically expire within 30 minutes of creation. Attempting to use expired collection tokens results in cryptic error messages and payment failures. Always convert collection tokens to payment method tokens for storage, or implement proper token refresh logic with clear expiry handling.

⚠ Pitfall: Mixing Test and Production Tokens Token formats appear identical between test and production environments, making it easy to accidentally submit test tokens to production payment endpoints or vice versa. This mistake causes confusing authorization failures that appear as customer card problems. Always validate token prefixes against expected environment indicators, maintain separate database schemas for test vs production tokens, and implement environment consistency checks in API request validation.

⚠ Pitfall: Insufficient Token Metadata Validation Relying solely on tokenization for security while ignoring token metadata validation creates vulnerabilities around duplicate payment methods and customer identity verification. Attackers may create multiple tokens for the same card or use valid tokens with incorrect customer associations. Always validate token fingerprints against existing customer payment methods, verify customer ownership before allowing token usage, and implement duplicate token detection based on card fingerprints rather than token IDs.

⚠ Pitfall: Inadequate Token Cleanup Unused payment method tokens accumulate over time, creating security liability and potential compliance issues during audits. Expired tokens, tokens from deleted customer accounts, and tokens that haven't been used for extended periods should be systematically purged. Implement automated token cleanup jobs that respect retention requirements, provide customer notification before removing saved payment methods, and maintain audit logs of token deletion activities for compliance purposes.

Implementation Guidance

The payment processing engine requires careful integration of external provider APIs, secure token handling, and complex state management for 3D Secure authentication flows. This implementation guidance provides concrete code structures and patterns to handle these requirements reliably.

A. Technology Recommendations:

Component	Simple Option	Advanced Option
HTTP Client	<code>requests</code> with connection pooling	<code>aiohttp</code> for async processing with connection management
Provider Integration	Direct API calls with <code>requests</code>	Provider SDK libraries (<code>stripe-python</code> , <code>adyen-python</code>)
Token Encryption	Provider-managed tokenization only	Additional field-level encryption using <code>cryptography</code>
State Management	Database records with status fields	Event sourcing with state reconstruction
3DS Redirect Handling	Simple URL generation with HMAC signatures	JWT tokens with embedded session state
Background Processing	<code>celery</code> task queue for async operations	<code>asyncio</code> with <code>aioredis</code> for real-time processing

B. Recommended File Structure:

```
payment-gateway/
  internal/
    processing/
      engine.py           ← Main processing engine class
      providers/
        base.py          ← Abstract provider interface
        stripe_provider.py   ← Stripe API implementation
        adyen_provider.py   ← Adyen API implementation
      tokenization.py     ← Token management utilities
      three_ds.py         ← 3D Secure flow handling
      charge_processor.py  ← Core charge processing logic
    models/
      charge.py          ← Charge database model
      payment_method_token.py  ← Token storage model
    api/
      payment_routes.py    ← HTTP endpoints for payment processing
      webhook_routes.py    ← Provider webhook handlers
  tests/
    test_processing_engine.py  ← Core engine tests
    test_three_ds_flow.py     ← 3DS authentication tests
    test_provider_integration.py  ← Provider API integration tests
```

C. Infrastructure Starter Code:

Complete payment provider abstraction interface:

```
from abc import ABC, abstractmethod

from typing import Dict, Optional, Any

from dataclasses import dataclass

from enum import Enum


class AuthenticationStatus(Enum):

    NOT_REQUIRED = "not_required"

    REQUIRED = "required"

    COMPLETED = "completed"

    FAILED = "failed"


@dataclass

class ChargeResponse:

    provider_charge_id: str

    status: str

    authentication_url: Optional[str] = None

    client_secret: Optional[str] = None

    failure_code: Optional[str] = None

    failure_message: Optional[str] = None

    network_transaction_id: Optional[str] = None

    authorization_code: Optional[str] = None


@dataclass

class AuthenticationResult:

    status: AuthenticationStatus

    authentication_token: Optional[str] = None

    failure_reason: Optional[str] = None


class PaymentProvider(ABC):

    """Abstract interface for payment provider integrations."""

    @abstractmethod
```

```
async def create_charge(
    self,
    payment_method_token: str,
    amount: int,
    currency: str,
    return_url: str,
    metadata: Dict[str, Any]
) -> ChargeResponse:

    """Create a new charge with the provider."""
    pass


@abstractmethod

async def retrieve_charge(self, provider_charge_id: str) -> ChargeResponse:

    """Retrieve current charge status from provider."""
    pass


@abstractmethod

async def confirm_payment(
    self,
    provider_charge_id: str,
    authentication_token: str
) -> ChargeResponse:

    """Confirm payment after 3DS authentication."""
    pass


@abstractmethod

async def cancel_charge(self, provider_charge_id: str) -> ChargeResponse:

    """Cancel a pending charge before settlement."""
    pass
```

```
class CircuitBreaker:

    """Circuit breaker for provider API calls."""

    def __init__(self, failure_threshold: int = 5, timeout_seconds: int = 60):

        self.failure_threshold = failure_threshold

        self.timeout_seconds = timeout_seconds

        self.failure_count = 0

        self.last_failure_time = None

        self.state = "closed" # closed, open, half_open

    @asyncio.coroutine
    def call(self, func, *args, **kwargs):

        if self.state == "open":

            if self._should_attempt_reset():

                self.state = "half_open"

            else:

                raise Exception("Circuit breaker is open")

        try:

            result = await func(*args, **kwargs)

            self._on_success()

            return result

        except Exception as e:

            self._on_failure()

            raise

    def _should_attempt_reset(self) -> bool:

        if self.last_failure_time is None:

            return True

        import time

        return time.time() - self.last_failure_time > self.timeout_seconds
```

```
def _on_success(self):

    self.failure_count = 0

    self.state = "closed"

def _on_failure(self):

    self.failure_count += 1

    import time

    self.last_failure_time = time.time()

    if self.failure_count >= self.failure_threshold:

        self.state = "open"
```

3D Secure URL generation and validation utilities:

```
import hmac

import hashlib

import time

import secrets

from urllib.parse import urlencode, parse_qs

from typing import Dict, Optional


class ThreeDSUrlManager:

    """Manages 3D Secure return URL generation and validation."""

    def __init__(self, secret_key: str, base_return_url: str):

        self.secret_key = secret_key.encode('utf-8')

        self.base_return_url = base_return_url


    def generate_return_url(

        self,

        payment_intent_id: str,

        session_id: str,

        expires_in_minutes: int = 30

    ) -> str:

        """Generate secure return URL for 3DS authentication."""

        timestamp = int(time.time())

        expiry = timestamp + (expires_in_minutes * 60)


        params = {

            'payment_intent_id': payment_intent_id,

            'session_id': session_id,

            'timestamp': str(timestamp),

            'expires': str(expiry)

        }
```

```
# Create HMAC signature of all parameters

param_string = '&'.join(f'{k}={v}' for k, v in sorted(params.items()))

signature = hmac.new(
    self.secret_key,
    param_string.encode('utf-8'),
    hashlib.sha256
).hexdigest()

params['signature'] = signature

return f"{self.base_return_url}?{urlencode(params)}"

def validate_return_params(self, query_params: Dict[str, str]) -> bool:

    """Validate 3DS return URL parameters and signature."""

    required_params = ['payment_intent_id', 'session_id', 'timestamp', 'expires', 'signature']

    if not all(param in query_params for param in required_params):

        return False

    # Check expiry

    current_time = int(time.time())

    if current_time > int(query_params['expires']):

        return False

    # Verify signature

    signature = query_params.pop('signature')

    param_string = '&'.join(f'{k}={v}' for k, v in sorted(query_params.items()))

    expected_signature = hmac.new(
        self.secret_key,
        param_string.encode('utf-8'),
        hashlib.sha256
    ).hexdigest()

    if signature != expected_signature:

        return False

    return True
```

```
    ).hexdigest()

    return hmac.compare_digest(signature, expected_signature)

def generate_session_id() -> str:
    """Generate cryptographically secure session ID."""
    return f"3ds_sess_{secrets.token_urlsafe(32)}"
```

D. Core Logic Skeleton Code:

```
from typing import Optional, Dict, Any

from dataclasses import asdict

import logging

from sqlalchemy.orm import Session

class PaymentProcessingEngine:

    """Core payment processing engine handling charge creation and 3DS flows."""

    def __init__(

        self,
        provider: PaymentProvider,
        circuit_breaker: CircuitBreaker,
        three_ds_manager: ThreeDSUrlManager,
        logger: logging.Logger
    ):

        self.provider = provider
        self.circuit_breaker = circuit_breaker
        self.three_ds_manager = three_ds_manager
        self.logger = logger

    async def process_payment_intent(
        self,
        payment_intent_id: str,
        payment_method_token: str,
        db: Session
    ) -> Charge:

        """Process a payment intent by creating and executing a charge."""

        # TODO 1: Retrieve payment intent from database and validate it's in 'created' status
        # TODO 2: Validate payment method token exists and is not expired
        # TODO 3: Update payment intent status to 'processing'
```

```

# TODO 4: Create charge record in 'pending' status with payment intent reference

# TODO 5: Generate 3DS return URL using payment intent ID and unique session

# TODO 6: Use circuit breaker to call provider.create_charge() with token and return URL

# TODO 7: Update charge record with provider response (charge ID, status, auth URL)

# TODO 8: If 3DS required, update payment intent status to 'requires_action'

# TODO 9: If charge succeeded immediately, update payment intent status to 'succeeded'

# TODO 10: If charge failed, update both charge and payment intent to failed status

# TODO 11: Commit database transaction and return updated charge record

# Hint: Use database transactions to ensure consistent state updates

# Hint: Log all provider interactions for debugging and audit purposes

pass

async def handle_three_ds_return(
    self,
    query_params: Dict[str, str],
    db: Session
) -> Charge:

    """Handle customer return from 3D Secure authentication."""

    # TODO 1: Validate return URL parameters and signature using three_ds_manager

    # TODO 2: Extract payment_intent_id and session_id from validated parameters

    # TODO 3: Retrieve payment intent and verify it's in 'requires_action' status

    # TODO 4: Find associated charge record in 'pending' status

    # TODO 5: Extract authentication token/result from provider-specific parameters

    # TODO 6: Use circuit breaker to call provider.confirm_payment() with auth token

    # TODO 7: Update charge record with final status from provider response

    # TODO 8: Update payment intent status based on charge result (succeeded/failed)

    # TODO 9: If payment succeeded, capture authorization code and network transaction ID

    # TODO 10: Commit database transaction and return updated charge record

    # Hint: Authentication failures should update status to 'failed', not raise exceptions

    # Hint: Always validate session hasn't been used before to prevent replay attacks

```

```
pass

async def retry_failed_charge(
    self,
    original_charge_id: str,
    db: Session
) -> Charge:

    """Retry a failed charge by creating a new charge attempt."""

    # TODO 1: Retrieve original charge and validate it's in 'failed' status

    # TODO 2: Check failure_code to determine if retry is appropriate (not for permanent failures)

    # TODO 3: Retrieve associated payment intent and reset status to 'processing'

    # TODO 4: Create new charge record linked to same payment intent

    # TODO 5: Use same payment method token as original charge

    # TODO 6: Process new charge using normal flow (may require 3DS again)

    # TODO 7: Update payment intent status based on retry result

    # Hint: Some failure codes (expired card, insufficient funds) should not be retried

    # Hint: Implement exponential backoff for automatic retries to avoid provider rate limits

    pass

class TokenizationManager:

    """Manages payment method tokenization and lifecycle."""

    def __init__(self, provider: PaymentProvider):
        self.provider = provider

    async def create_payment_method_token(
        self,
        collection_token: str,
        customer_id: str,
        db: Session
    ) -> PaymentMethodToken:
```

```

) -> PaymentMethodToken:

    """Convert collection token to permanent payment method token."""

    # TODO 1: Validate collection token format and ensure it hasn't been used before

    # TODO 2: Call provider API to convert collection token to payment method token

    # TODO 3: Extract card metadata (last 4 digits, brand, expiry) from provider response

    # TODO 4: Check for duplicate payment methods using card fingerprint

    # TODO 5: Create PaymentMethodToken record with metadata and customer association

    # TODO 6: Mark collection token as used to prevent reuse

    # TODO 7: Store token with appropriate expiry based on card expiry date

    # TODO 8: Commit database transaction and return payment method token

    # Hint: Use card fingerprints to detect when customer adds same card multiple times

    # Hint: Store minimal metadata - never store actual card numbers or CVV

    pass

def validate_token_ownership(
    self,
    token_id: str,
    customer_id: str,
    db: Session
) -> bool:

    """Verify customer owns the specified payment method token."""

    # TODO 1: Retrieve payment method token from database by token_id

    # TODO 2: Compare token's customer_id with provided customer_id

    # TODO 3: Check token hasn't expired based on card expiry date

    # TODO 4: Return True only if customer owns token and it's still valid

    # Hint: This prevents customers from using other customers' payment methods

    pass

```

E. Language-Specific Hints:

- Use `asyncio` and `aiohttp` for non-blocking provider API calls to improve performance under load
- Implement proper connection pooling with `aiohttp.TCPConnector(limit=100)` to reuse HTTP connections

- Use `sqlalchemy.orm.sessionmaker` with `expire_on_commit=False` for database sessions in async contexts
- Store sensitive configuration like provider API keys in environment variables, never in code
- Use `logging.getLogger(__name__)` for structured logging with correlation IDs for request tracing
- Implement request timeouts with `asyncio.wait_for()` to prevent hanging on slow provider responses
- Use `secrets.token_urlsafe()` for generating cryptographically secure session IDs and tokens
- Handle provider API rate limits with `asyncio.sleep()` and exponential backoff patterns

F. Milestone Checkpoint:

After implementing the payment processing engine, verify the following functionality:

Test Commands:

```
# Run core processing engine tests
python -m pytest tests/test_processing_engine.py -v

# Test 3D Secure flow with mock provider
python -m pytest tests/test_three_ds_flow.py::test_authentication_redirect -v

# Integration test with test provider credentials
python -m pytest tests/test_provider_integration.py --integration
```

BASH

Expected Behavior:

1. **Charge Creation:** Submit a payment intent for processing and verify charge record is created with `pending` status
2. **3DS Challenge:** For test cards requiring authentication, verify payment intent transitions to `requires_action` with valid authentication URL
3. **Authentication Return:** Process 3DS return parameters and verify charge transitions to `succeeded` status
4. **Token Validation:** Attempt to use invalid or expired tokens and verify appropriate error responses
5. **Circuit Breaker:** Simulate provider failures and verify circuit breaker opens after threshold failures

Manual Verification:

- Start the payment server and create a payment intent through the API
- Submit a charge using test card `400000000003220` (requires 3DS) and verify redirect URL is returned
- Complete authentication in test environment and verify return URL processes correctly
- Check database records show proper state transitions and provider response data
- Verify failed payments update charge records with appropriate error codes and messages

Signs of Problems:

- Charges stuck in `pending` status indicate provider communication issues
- Missing authentication URLs suggest 3DS integration problems
- Database constraint violations indicate improper transaction handling
- Signature validation failures on return URLs suggest HMAC implementation issues

Refunds and Dispute Management

Milestone(s): Milestone 3: Refunds & Disputes - implementing refund processing, partial refunds, and dispute handling with proper state tracking and chargeback workflows.

Think of refunds and disputes as the "undo" and "investigation" mechanisms of the payment world. When you return an item to a store, the clerk processes a refund by reversing part or all of your original transaction - this happens through the same card network that processed your payment, but in reverse. Disputes are like filing a complaint with your credit card company when something goes wrong - the card issuer investigates and may force the merchant to return your money, often before the merchant even knows there's a problem. Our system needs to handle both scenarios gracefully while maintaining accurate accounting records.

The complexity comes from the asynchronous nature of these operations. Unlike the immediate feedback you get when making a purchase, refunds can take days to appear on a customer's statement, and disputes can take weeks or months to resolve. Meanwhile, money moves through multiple parties - your business, the payment processor, the card networks, and various banks - each maintaining their own records that must eventually reconcile.

Refund Processing Logic

Refunds represent the reversal of value transfer from a completed charge. Unlike voiding a transaction (which cancels it before settlement), refunds actually move money backward through the payment network. This fundamental difference affects timing, fees, and the customer experience.

The refund state machine mirrors the charge lifecycle but with important distinctions. Refunds begin in a `pending` status while the payment processor validates the request and initiates the reversal. They transition to `succeeded` when the processor confirms the refund will be sent to the customer's account, or `failed` if validation fails or the refund is rejected. The time between initiation and customer visibility varies significantly - typically 3-10 business days for card refunds.

Current State	Event	Next State	Actions Taken
<code>pending</code>	<code>processor_confirms</code>	<code>succeeded</code>	Update network transaction ID, set <code>processed_at</code> timestamp
<code>pending</code>	<code>processor_rejects</code>	<code>failed</code>	Record failure code and message, alert operations team
<code>succeeded</code>	<code>none</code>	<code>terminal</code>	No further state changes allowed
<code>failed</code>	<code>none</code>	<code>terminal</code>	No further state changes allowed

Partial refunds introduce additional validation complexity. The system must track the cumulative refunded amount against the original charge to prevent over-refunding. This requires maintaining a running total of successful refunds and comparing it against the original charge amount before approving new refund requests.

Decision: Refund Amount Validation Strategy

- **Context:** Partial refunds require preventing over-refunding while allowing multiple small refunds against a single charge
- **Options Considered:**
 - Pre-calculate remaining refundable amount and store it
 - Calculate refundable amount dynamically by summing existing refunds
 - Rely on payment processor to validate refund amounts
- **Decision:** Calculate refundable amount dynamically by summing existing refunds
- **Rationale:** Avoids data synchronization issues and provides authoritative validation using the database as single source of truth
- **Consequences:** Requires database query on each refund but ensures accuracy and simplifies concurrent refund handling

The refund validation algorithm follows these steps:

1. **Load the original charge** using the provided charge ID and verify it exists in `succeeded` status
2. **Query all existing refunds** for this charge that have `succeeded` status
3. **Calculate the total refunded amount** by summing the amounts from all successful refunds
4. **Determine the remaining refundable amount** by subtracting total refunded from original charge amount
5. **Validate the requested refund amount** is positive and does not exceed the remaining refundable amount
6. **Check charge age constraints** - most processors limit refunds to charges made within 180 days
7. **Verify charge settlement status** - some processors require charges to be fully settled before allowing refunds

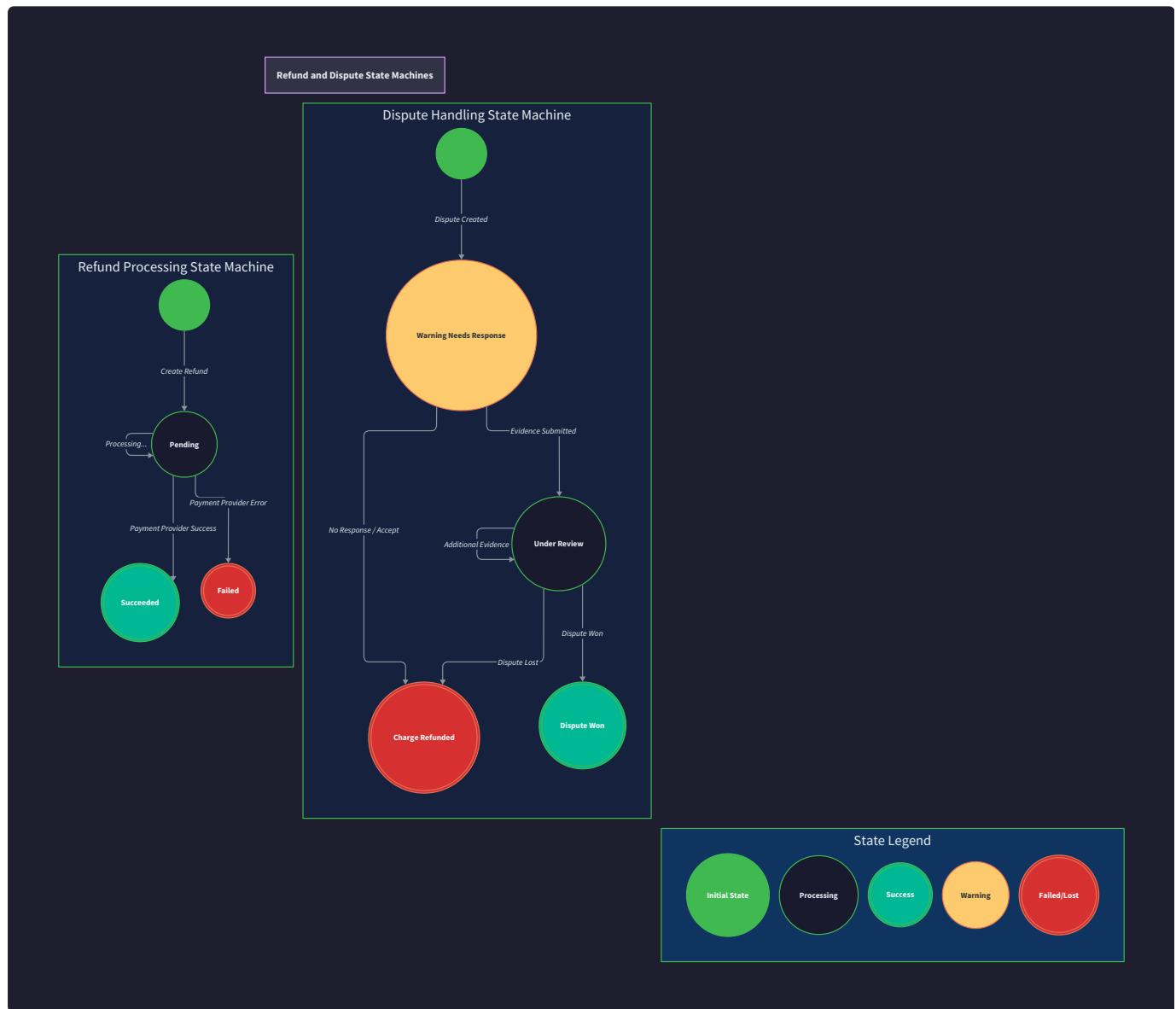
Refund Field	Type	Description
<code>id</code>	<code>str</code>	Unique refund identifier generated by our system
<code>charge_id</code>	<code>str</code>	Foreign key reference to the original charge being refunded
<code>amount</code>	<code>int</code>	Refund amount in smallest currency unit (cents)
<code>status</code>	<code>RefundStatus</code>	Current refund state: pending, succeeded, failed
<code>reason</code>	<code>str</code>	Merchant-provided reason for the refund (fraud, return, duplicate, etc.)
<code>network_transaction_id</code>	<code>str</code>	Payment processor's refund transaction identifier
<code>failure_code</code>	<code>str</code>	Processor-specific error code if refund failed
<code>failure_message</code>	<code>str</code>	Human-readable failure description
<code>created_at</code>	<code>datetime</code>	Timestamp when refund was initiated
<code>processed_at</code>	<code>datetime</code>	Timestamp when processor confirmed refund completion

Refund reasons serve both operational and compliance purposes. Common reason codes include "duplicate" for accidentally charged transactions, "fraudulent" for confirmed fraud cases, "requested_by_customer" for returns and

cancellations, and "expired_uncaptured_charge" for authorization reversals. These reasons help with reconciliation, dispute prevention, and regulatory reporting.

The interaction between refunds and the original payment method adds complexity. Card refunds must go back to the original card - you cannot refund a Visa charge to a Mastercard or to a bank account. If the original card has been closed or replaced, the refund may fail, requiring alternative resolution methods like check payments or store credit.

Critical Insight: Refund Idempotency Refund requests must be idempotent to prevent duplicate refunds from UI retries or API client errors. Include an idempotency key parameter in refund creation requests and store it alongside the refund record to detect duplicates.



Dispute and Chargeback Workflow

Disputes represent forced payment reversals initiated by the customer's bank rather than the merchant. Think of this as the difference between voluntarily returning an item versus filing a complaint with customer service that forces a refund. The merchant has limited control over the process and faces strict deadlines for response.

The dispute lifecycle begins when a cardholder contacts their bank claiming a problem with a transaction. The bank may issue a chargeback immediately (taking money from the merchant account) or send a warning that gives the merchant a chance to resolve the issue first. This early warning system, sometimes called "pre-arbitration," allows merchants to voluntarily refund disputed transactions before facing forced chargebacks.

Dispute Status	Description	Merchant Action Required	Timeline
warning_needs_response	Bank inquiry before formal chargeback	Respond with refund or evidence	7-10 days
under_review	Formal chargeback filed, evidence submitted	Wait for network decision	30-75 days
charge_refunded	Merchant voluntarily refunded during warning	None	Terminal
won	Evidence accepted, funds returned to merchant	None	Terminal
lost	Evidence rejected, chargeback upheld	Accept loss or appeal	May appeal

Chargeback reason codes determine the appropriate response strategy. Fraud-related chargebacks (reason code 10.4 for Visa) require different evidence than authorization disputes (code 11.1) or processing errors (code 12.1). Understanding these codes allows automated evidence collection and response routing.

Common reason categories include:

- **Fraud:** Customer claims they didn't make the transaction
- **Authorization:** Transaction processed without proper authorization
- **Processing Errors:** Duplicate processing, wrong amount, or currency issues
- **Consumer Disputes:** Product not received, defective, or not as described

Decision: Automated vs Manual Dispute Response

- **Context:** Disputes require evidence submission within tight deadlines but manual review ensures quality
- **Options Considered:**
 - Fully automated evidence submission based on reason codes
 - Manual review for all disputes with staff assignment
 - Hybrid approach with automation for simple cases, manual for complex
- **Decision:** Hybrid approach with automation for fraud and authorization disputes, manual review for consumer disputes
- **Rationale:** Fraud disputes have standardized evidence requirements while consumer disputes need contextual judgment about product/service delivery
- **Consequences:** Reduces response time for high-volume fraud disputes while maintaining quality for complex cases

The evidence collection system must gather relevant data based on the dispute type. For fraud disputes, this includes customer authentication records, device fingerprinting data, shipping confirmations, and previous successful transactions.

For authorization disputes, the focus shifts to AVS (Address Verification Service) responses, CVV matches, and authorization codes.

Evidence Type	Fraud Disputes	Authorization Disputes	Consumer Disputes
Customer authentication	Required	Optional	Optional
Shipping/delivery proof	Required	Not applicable	Required
Product description	Optional	Not applicable	Required
Customer communication	Optional	Optional	Required
Authorization codes	Optional	Required	Optional
Device fingerprint	Required	Optional	Not applicable

Dispute fees apply regardless of outcome. Most payment processors charge \$15-25 per dispute even if the merchant ultimately wins. This creates a strong incentive for dispute prevention through clear billing descriptors, proactive customer service, and fraud prevention systems.

The automated evidence submission system follows this workflow:

1. **Receive dispute notification** via webhook from payment processor
2. **Parse reason code and dispute details** to determine evidence requirements
3. **Query relevant data sources** including order systems, shipping providers, and authentication logs
4. **Compile evidence package** according to card network specifications
5. **Submit evidence** through payment processor API before deadline
6. **Update dispute status** and schedule follow-up for network decision
7. **Handle dispute resolution** by updating accounting records and notifying relevant systems

⚠ Pitfall: Missing Evidence Deadlines Dispute evidence deadlines are strictly enforced - missing a deadline by even minutes results in automatic loss. Implement alert systems that notify operations teams 48-72 hours before deadlines and automate evidence submission for standard cases. Never rely on manual processes for time-critical compliance requirements.

Payment Accounting Ledger

Payment accounting requires double-entry bookkeeping to maintain accurate records across refunds, disputes, and fees. Think of this as maintaining a detailed bank register that tracks every penny moving in and out of your business, with each transaction affecting multiple accounts simultaneously.

Traditional payment systems often use simple status fields and running balances, but this approach breaks down with complex scenarios like partial refunds, dispute fees, and chargeback reversals. A proper ledger system records each financial event as journal entries affecting multiple accounts, providing an audit trail and ensuring mathematical consistency.

The core insight is that every financial event affects at least two accounts. When processing a \$100 payment, we debit "Customer Payments Receivable" for \$100 and credit "Sales Revenue" for \$100. When refunding \$30 of that payment, we debit "Refunds Expense" for \$30 and credit "Customer Refunds Payable" for \$30. This dual-entry approach makes discrepancies immediately visible and supports complex reconciliation scenarios.

Account Type	Account Name	Debit Increases	Credit Increases	Example Transactions
Asset	Customer Payments Receivable	Yes	No	Original payment processing
Liability	Customer Refunds Payable	No	Yes	Refund obligations to customers
Expense	Processing Fees	Yes	No	Payment processor fees
Expense	Dispute Fees	Yes	No	Chargeback and dispute fees
Expense	Refunds Expense	Yes	No	Customer refunds processed
Revenue	Sales Revenue	No	Yes	Completed customer purchases

Journal entries capture the complete financial impact of each payment event. A simple payment involves multiple fees and timing considerations that must be recorded accurately. Consider a \$100 payment with a \$2.90 processing fee:

Payment Processing Entry:

- Debit Customer Payments Receivable: \$100.00
- Credit Sales Revenue: \$100.00
- Debit Processing Fees: \$2.90
- Credit Processing Fees Payable: \$2.90

Later, when issuing a \$30 refund with a \$0.50 refund fee:

Refund Processing Entry:

- Debit Refunds Expense: \$30.00
- Credit Customer Refunds Payable: \$30.00
- Debit Refund Fees: \$0.50
- Credit Refund Fees Payable: \$0.50

Dispute accounting creates additional complexity because chargebacks represent involuntary fund movements.

When a dispute is filed, the payment processor immediately debits the chargeback amount from the merchant account, creating a negative balance that must be tracked separately from voluntary refunds.

Chargeback Journal Entry:

- Debit Chargebacks Expense: \$100.00
- Credit Chargebacks Payable: \$100.00
- Debit Dispute Fees: \$15.00
- Credit Dispute Fees Payable: \$15.00

If the merchant wins the dispute, the chargeback reverses:

Chargeback Reversal Entry:

- Debit Chargebacks Receivable: \$100.00
- Credit Chargebacks Expense: \$100.00

Decision: Ledger Entry Timing Strategy

- Context:** Journal entries must be recorded when financial events occur, but payment events are asynchronous
- Options Considered:**
 - Record entries when initiating actions (payment requests, refund requests)
 - Record entries when receiving confirmations (webhooks, API responses)
 - Record provisional entries immediately, then confirm or reverse based on outcomes
- Decision:** Record provisional entries immediately, then confirm or reverse based on outcomes
- Rationale:** Provides immediate financial visibility for operational decisions while maintaining accuracy through reconciliation
- Consequences:** Requires more complex ledger logic but gives real-time financial position and audit trail

The reconciliation process ensures ledger accuracy by comparing internal records against payment processor settlements. Payment processors provide daily settlement reports showing actual fund movements, which must match the sum of confirmed ledger entries.

Reconciliation Field	Internal Calculation	External Source	Variance Threshold
Gross Sales	Sum of confirmed sales revenue entries	Processor gross sales	\$0.01
Total Refunds	Sum of confirmed refund expense entries	Processor total refunds	\$0.01
Processing Fees	Sum of confirmed fee entries	Processor fee report	\$0.05
Net Settlement	Gross - Refunds - Fees	Processor net deposit	\$0.10

Settlement timing differences require careful handling. Payment processors typically settle funds 1-2 business days after transaction processing, but the exact timing varies by transaction type, risk assessment, and processor policies. The ledger must track these timing differences to provide accurate cash flow projections.

⚠ Pitfall: Currency Precision Errors Store all monetary amounts as integers in the smallest currency unit (cents for USD) to avoid floating-point precision errors. A \$10.00 payment is stored as 1000 cents. This is critical for ledger accuracy - even one-cent discrepancies multiply across thousands of transactions and cause reconciliation failures.

The ledger query interface provides standard reports for business operations:

Report Type	Query Logic	Use Case
Daily Sales	Sum confirmed sales entries by date	Revenue reporting
Refund Analysis	Sum refunds by reason and date range	Return pattern analysis
Dispute Impact	Sum dispute fees and chargebacks	Loss prevention metrics
Settlement Forecast	Sum pending entries by expected settlement date	Cash flow planning

Implementation Guidance

A. Technology Recommendations:

Component	Simple Option	Advanced Option
Refund Processing	HTTP REST API with JSON	Event-driven with message queues
Ledger Storage	PostgreSQL with ACID transactions	Dedicated ledger database (e.g., TigerBeetle)
Evidence Collection	File storage with metadata	Document management system
Reconciliation	Scheduled batch jobs	Real-time streaming reconciliation

B. Recommended File Structure:

```

internal/refunds/
    refund_service.go      ← Main refund processing logic
    refund_validator.go    ← Amount and constraint validation
    refund_repository.go   ← Database operations
    refund_test.go         ← Refund processing tests

internal/disputes/
    dispute_service.go     ← Dispute handling workflow
    evidence_collector.go  ← Automated evidence gathering
    dispute_repository.go  ← Dispute data persistence
    dispute_test.go        ← Dispute workflow tests

internal/ledger/
    ledger_service.go      ← Double-entry bookkeeping logic
    journal_entry.go        ← Journal entry data structures
    reconciliation_service.go ← Settlement reconciliation
    ledger_test.go          ← Accounting accuracy tests

internal/webhooks/
    dispute_webhook_handler.go ← Chargeback notification processing

```

C. Infrastructure Starter Code:

```
# ledger/journal_entry.py - Complete journal entry infrastructure

from dataclasses import dataclass

from datetime import datetime

from decimal import Decimal

from enum import Enum

from typing import List, Optional


class AccountType(Enum):

    ASSET = "asset"

    LIABILITY = "liability"

    EXPENSE = "expense"

    REVENUE = "revenue"

    @dataclass

    class LedgerAccount:

        """Represents a general ledger account."""

        id: str

        name: str

        account_type: AccountType

        normal_balance: str # "debit" or "credit"

        def is_debit_increase(self) -> bool:

            """Returns True if debits increase this account balance."""

            return self.account_type in [AccountType.ASSET, AccountType.EXPENSE]

    @dataclass

    class JournalEntryLine:

        """Individual line item in a journal entry."""

        account_id: str

        debit_amount: Optional[int] # Amount in cents

        credit_amount: Optional[int] # Amount in cents
```

```
description: str

def validate(self) -> bool:
    """Validates that exactly one of debit or credit is set."""
    return bool(self.debit_amount) != bool(self.credit_amount)

@dataclass
class JournalEntry:

    """Complete journal entry with multiple line items."""

    id: str
    reference_type: str # "payment", "refund", "dispute"
    reference_id: str # ID of the related payment/refund/dispute
    description: str
    lines: List[JournalEntryLine]
    created_at: datetime
    posted_at: Optional[datetime]

    def validate(self) -> bool:
        """Validates that debits equal credits."""
        total_debits = sum(line.debit_amount or 0 for line in self.lines)
        total_credits = sum(line.credit_amount or 0 for line in self.lines)
        return total_debits == total_credits and len(self.lines) >= 2

class LedgerService:

    """Service for recording journal entries and maintaining ledger."""

    def __init__(self, db_session):
        self.db = db_session
        self._setup_standard_accounts()

    def _setup_standard_accounts(self):
```

```

"""Initialize standard chart of accounts."""

standard_accounts = [
    LedgerAccount("1000", "Customer Payments Receivable", AccountType.ASSET, "debit"),
    LedgerAccount("2000", "Customer Refunds Payable", AccountType.LIABILITY, "credit"),
    LedgerAccount("4000", "Sales Revenue", AccountType.REVENUE, "credit"),
    LedgerAccount("5000", "Refunds Expense", AccountType.EXPENSE, "debit"),
    LedgerAccount("5100", "Processing Fees", AccountType.EXPENSE, "debit"),
    LedgerAccount("5200", "Dispute Fees", AccountType.EXPENSE, "debit"),
    LedgerAccount("5300", "Chargebacks Expense", AccountType.EXPENSE, "debit"),
]

# Implementation: Store accounts in database


def post_journal_entry(self, entry: JournalEntry) -> bool:
    """Posts a validated journal entry to the ledger."""

    if not entry.validate():
        raise ValueError(f"Invalid journal entry: {entry}")

    # Implementation: Insert entry and lines in database transaction

    # Set posted_at timestamp

    # Update account balances

    pass


# Evidence collection infrastructure

class EvidenceCollector:

    """Collects dispute evidence from various data sources."""

    def __init__(self, shipping_api, auth_service, order_service):
        self.shipping = shipping_api
        self.auth = auth_service
        self.orders = order_service

```

```
def collect_fraud_evidence(self, charge_id: str) -> dict:

    """Collects evidence package for fraud disputes."""

    evidence = {}

    # Collect authentication evidence

    auth_data = self.auth.get_authentication_data(charge_id)

    evidence["customer_authentication"] = {

        "method": auth_data.method,

        "timestamp": auth_data.timestamp,

        "ip_address": auth_data.ip_address,

        "device_fingerprint": auth_data.device_id

    }

    # Collect shipping evidence

    shipping_data = self.shipping.get_delivery_confirmation(charge_id)

    if shipping_data:

        evidence["delivery_confirmation"] = {

            "tracking_number": shipping_data.tracking_number,

            "delivery_date": shipping_data.delivered_at,

            "delivery_address": shipping_data.address,

            "signature_required": shipping_data.signature_required

        }

    return evidence
```

D. Core Logic Skeleton Code:

```
# refunds/refund_service.py - Core refund processing logic
```

PYTHON

```
class RefundService:

    """Handles refund creation and processing workflow."""

    def __init__(self, db_session, payment_provider, ledger_service):

        self.db = db_session

        self.provider = payment_provider

        self.ledger = ledger_service

    def create_refund(self, charge_id: str, amount: int, reason: str,
                      idempotency_key: str) -> Refund:

        """Creates and processes a refund against a completed charge.

        Args:

            charge_id: ID of the original charge to refund

            amount: Refund amount in cents (must be <= remaining refundable)

            reason: Merchant reason for refund (affects dispute prevention)

            idempotency_key: Prevents duplicate refund processing

        Returns:

            Refund object in pending status

        Raises:

            ValueError: If amount exceeds refundable balance

            IntegrityError: If idempotency key was used with different params

        """

        # TODO 1: Check idempotency key for duplicate requests

        # TODO 2: Load original charge and validate it's in succeeded status

        # TODO 3: Calculate total previously refunded amount from database

        # TODO 4: Validate requested amount doesn't exceed refundable balance
```

Args:

```
charge_id: ID of the original charge to refund

amount: Refund amount in cents (must be <= remaining refundable)

reason: Merchant reason for refund (affects dispute prevention)

idempotency_key: Prevents duplicate refund processing
```

Returns:

```
Refund object in pending status
```

Raises:

```
ValueError: If amount exceeds refundable balance
```

```
IntegrityError: If idempotency key was used with different params
```

"""

```
# TODO 1: Check idempotency key for duplicate requests
```

```
# TODO 2: Load original charge and validate it's in succeeded status
```

```
# TODO 3: Calculate total previously refunded amount from database
```

```
# TODO 4: Validate requested amount doesn't exceed refundable balance
```


Args:

```
    webhook_data: Provider webhook with dispute details
```

Returns:

```
    True if dispute was processed successfully
```

```
"""
```

```
# TODO 1: Parse dispute data including reason code and deadline
```

```
# TODO 2: Create dispute record in appropriate status
```

```
# TODO 3: Record chargeback journal entry if funds were debited
```

```
# TODO 4: Determine if this is a warning or formal chargeback
```

```
# TODO 5: Route to automated or manual evidence collection
```

```
# TODO 6: Set up deadline alerts for evidence submission
```

```
# TODO 7: Notify operations team of dispute requiring action
```

```
pass
```

```
def submit_dispute_evidence(self, dispute_id: str, evidence: dict) -> bool:
```

```
    """Submits compiled evidence package for dispute resolution.
```

Args:

```
    dispute_id: Internal dispute identifier
```

```
    evidence: Evidence package formatted for card network
```

Returns:

```
    True if evidence was submitted successfully
```

```
"""
```

```
# TODO 1: Load dispute record and verify it's in appropriate status
```

```
# TODO 2: Validate evidence package contains required elements
```

```
# TODO 3: Submit evidence to payment provider before deadline
```

```
# TODO 4: Update dispute status to under_review
```

```
# TODO 5: Record evidence submission timestamp and provider response
```

```
# TODO 6: Schedule follow-up task to check for resolution  
pass
```

E. Language-Specific Hints:

- Use `decimal.Decimal` for currency calculations to avoid floating-point errors
- Implement database transactions with proper rollback for failed refund attempts
- Use `datetime.utcnow()` for all timestamps to avoid timezone confusion
- Store monetary amounts as integers (cents) in database schemas
- Use enum classes for status values to prevent invalid state assignments
- Implement webhook signature validation before processing any dispute notifications

F. Milestone Checkpoint:

After implementing refund processing:

1. Create a successful payment intent and charge through your API
2. Issue a partial refund for half the charge amount: `POST /refunds {"charge_id": "ch_xxx", "amount": 500, "reason": "requested_by_customer"}`
3. Verify refund appears in `pending` status with correct remaining refundable amount
4. Simulate webhook confirmation and verify refund transitions to `succeeded`
5. Attempt to refund more than the remaining balance and verify it's rejected
6. Check that journal entries show correct debit to Refunds Expense and credit to Customer Refunds Payable

For dispute handling:

1. Simulate a dispute webhook notification with fraud reason code
2. Verify dispute record is created with correct deadline calculation
3. Check that evidence collection runs automatically for fraud disputes
4. Verify chargeback journal entry debits Chargebacks Expense
5. Submit evidence package and verify status updates to `under_review`

G. Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
Refund amount validation fails	Floating-point precision in amount calculation	Check if amounts stored as floats vs integers	Store all amounts as integers in cents
Journal entries don't balance	Missing fee entries or incorrect debit/credit	Sum debits and credits for each journal entry	Add validation that debits = credits before posting
Dispute evidence submission fails	Missing required fields or wrong format	Check provider API response for validation errors	Map evidence fields to exact provider requirements
Webhook processing is slow	Database queries in webhook handler	Profile webhook handler performance	Move heavy processing to background jobs
Reconciliation shows discrepancies	Timing differences or missing transactions	Compare transaction timestamps with settlement reports	Implement grace period for pending transactions

Webhook Processing and Reconciliation

Milestone(s): Milestone 4: Webhook Reconciliation - implementing reliable webhook processing with signature verification and reconciliation, receiving payment webhooks from external providers, updating local payment status records, verifying webhook payload signatures, and reconciling local payment records against expected provider state.

Webhooks in payment processing are like having a direct telephone line between your payment provider and your system. Instead of constantly calling the provider asking "has anything changed?" (polling), the provider calls you immediately when something important happens to a payment. Think of it as the difference between repeatedly asking a friend "are you there yet?" during a road trip versus having them text you when they arrive. Webhooks eliminate the delay and uncertainty of polling while ensuring your system stays synchronized with the provider's authoritative payment state.

However, this convenience comes with significant challenges. Unlike API calls that you initiate and control, webhooks arrive asynchronously from external systems over the public internet. This creates a fundamental trust problem: how do you know a webhook claiming to be from your payment provider actually came from them and wasn't forged by an attacker? Additionally, network reliability issues mean webhooks can be delivered multiple times, arrive out of order, or fail to arrive at all. Your system must handle these realities while maintaining accurate payment state.

The webhook processing architecture addresses three critical concerns: **authenticity** (is this webhook genuine?), **idempotency** (can we safely process it multiple times?), and **completeness** (did we miss any events?). This section explores how to build a robust webhook processing pipeline that maintains payment state consistency even under adverse network conditions and potential security threats.

Webhook Security and Verification

Webhook signatures serve as cryptographic proof that an incoming webhook payload was generated by your payment provider and has not been tampered with during transmission. Think of webhook signatures like the wax seal on official letters from medieval times - they provide both authentication (this came from the claimed sender) and integrity verification (the contents haven't been altered). Without signature verification, your webhook endpoint becomes a vulnerability that attackers can exploit to manipulate your payment state.

The signature verification process relies on **HMAC (Hash-based Message Authentication Code)** cryptography. When your payment provider sends a webhook, they combine the webhook payload with a secret key (that only you and they know) and generate a cryptographic hash. This hash is included in the webhook headers. Your system performs the same calculation using the received payload and your copy of the secret key. If the hashes match, you can be confident the webhook is authentic and unmodified.

Decision: HMAC-SHA256 for Webhook Signature Verification

- **Context:** Need to verify webhook authenticity and prevent spoofed payment notifications that could manipulate transaction state
- **Options Considered:**
 1. No signature verification (trust all incoming webhooks)
 2. HMAC-SHA256 with timestamp validation
 3. RSA digital signatures with certificate validation
- **Decision:** HMAC-SHA256 with timestamp validation
- **Rationale:** Provides strong cryptographic verification while remaining simple to implement. HMAC-SHA256 is the industry standard for webhook signatures (used by Stripe, PayPal, etc.). RSA signatures add complexity without meaningful security benefits for this use case. No verification is completely unacceptable for payment systems.
- **Consequences:** Requires secure webhook secret management and proper timestamp validation to prevent replay attacks. Enables confident processing of webhook events knowing they're authentic.

The webhook signature verification process follows a precise algorithm that must be implemented exactly as specified by your payment provider. Most providers include the signature in a header like `Stripe-Signature` or `X-Webhook-Signature`. The header typically contains multiple components: the timestamp when the webhook was sent, and one or more signature values computed using different versions of the signing algorithm.

Signature Component	Purpose	Example Value
Timestamp	Prevents replay attacks by limiting webhook validity window	<code>t=1634567890</code>
Primary Signature	HMAC-SHA256 hash of timestamp + payload using current secret	<code>v1=3f5b8a7c9d...</code>
Fallback Signature	Hash using previous secret during key rotation periods	<code>v0=7a9c3d5f8b...</code>

The signature verification algorithm proceeds through several critical steps that must be executed in the correct order. First, your system extracts the timestamp and signature components from the webhook headers. The timestamp serves dual purposes: it's part of the signed payload (preventing timestamp manipulation) and enables replay attack protection by rejecting webhooks older than a configured threshold (typically 5-10 minutes).

Next, your system reconstructs the signed payload exactly as the provider created it. This typically involves concatenating the timestamp, a delimiter character (usually a period), and the raw webhook payload body. It's crucial to use the raw, unmodified payload bytes - any JSON parsing, whitespace normalization, or character encoding changes will cause signature verification to fail.

Your system then computes the HMAC-SHA256 hash of this reconstructed payload using your webhook secret. The resulting hash must be compared with the signature from the webhook headers using a timing-safe comparison function to

prevent timing attacks. Standard string equality comparison can leak information about partial matches through execution time variations.

Critical Security Insight: Never use standard string comparison (`==`) for signature verification. Use a constant-time comparison function that always takes the same amount of time regardless of how many characters match. This prevents timing attacks where attackers could gradually deduce valid signatures by measuring response times.

Timestamp validation prevents replay attacks where an attacker captures a legitimate webhook and resends it later to trigger duplicate processing. Your system should reject webhooks with timestamps older than your configured tolerance window. However, be cautious about making this window too narrow - legitimate webhooks can be delayed by network issues, provider retries, or load balancer queuing.

Validation Check	Acceptance Criteria	Rejection Response
Timestamp Age	Within last 10 minutes	400 Bad Request with "Webhook too old"
Signature Format	Valid header format with <code>t=</code> and <code>v1=</code> components	400 Bad Request with "Invalid signature format"
Signature Match	HMAC matches computed value	401 Unauthorized with "Invalid signature"
Payload Size	Within reasonable limits (typically 1MB)	413 Payload Too Large

The webhook secret itself requires careful management. It should be stored securely (environment variables, secrets management systems) and never logged or included in error messages. Payment providers typically allow webhook secret rotation, which requires your system to support multiple secrets during transition periods. During rotation, providers may include signatures computed with both old and new secrets, allowing your system to verify using either one.

Rate limiting and IP allowlisting provide additional security layers. Payment providers typically send webhooks from specific IP address ranges that you can allowlist. However, these ranges can change, so IP allowlisting should supplement, not replace, signature verification. Rate limiting prevents abuse if your webhook endpoint is discovered by attackers, but be careful not to block legitimate webhook bursts during high-volume periods.

⚠ Pitfall: Signature Verification Bypass in Development Many developers disable signature verification in development environments for convenience, then accidentally deploy this configuration to production. This creates a critical security vulnerability where attackers can manipulate payment state by sending fake webhooks. Always enforce signature verification in all environments - use test webhook secrets in development, never skip verification entirely.

Event Processing Pipeline

The **event processing pipeline** transforms incoming webhook notifications into reliable updates to your payment system's state. Think of this pipeline like a factory assembly line for processing mail - incoming packages (webhooks) must be inspected for authenticity, sorted by type, checked for duplicates, processed according to their contents, and tracked to ensure nothing gets lost. Each stage has specific responsibilities and failure modes that must be handled gracefully.

The pipeline operates on a fundamental principle: **at-least-once delivery semantics**. Payment providers will retry failed webhook deliveries, potentially multiple times over several hours or days. This means your system will likely receive duplicate webhooks for the same event. Rather than trying to prevent duplicates (which is impossible to guarantee), your system must be **idempotent** - processing the same webhook multiple times should produce the same final state as processing it once.

Decision: Database-First Webhook Processing with Explicit Deduplication

- **Context:** Need to handle duplicate webhook deliveries reliably while ensuring payment state updates are never lost or applied incorrectly
- **Options Considered:**
 1. In-memory deduplication with immediate event processing
 2. Database-first with explicit deduplication table
 3. Message queue with exactly-once delivery guarantees
- **Decision:** Database-first with explicit deduplication table
- **Rationale:** Provides durability guarantees by persisting webhooks before processing. Database transactions ensure atomic deduplication and state updates. Message queues add complexity and external dependencies without solving the core idempotency challenge.
- **Consequences:** Requires careful transaction management and proper indexing for webhook deduplication. Enables reliable processing with clear audit trails of all webhook events.

The webhook processing flow begins with **webhook receipt and persistence**. When a webhook arrives at your endpoint, the first action is to persist it to your database with a unique identifier, timestamp, and processing status. This creates a permanent audit trail and ensures no webhook can be lost due to process crashes or network failures. The webhook record captures all essential metadata needed for processing and debugging.

WebhookEvent Field	Purpose	Example Value
<code>id</code>	Unique webhook identifier	<code>wh_1234567890abcdef</code>
<code>event_type</code>	Type of payment event	<code>payment_intent.succeeded</code>
<code>payload</code>	Complete webhook payload	<code>{"id": "pi_...", "status": "succeeded"}</code>
<code>signature</code>	Webhook signature for verification	<code>t=1634567890, v1=3f5b8a...</code>
<code>processed_at</code>	When processing completed successfully	<code>2023-10-15T14:30:00Z</code>
<code>processing_attempts</code>	Number of processing attempts	<code>2</code>
<code>last_processing_error</code>	Most recent processing error	<code>"Payment intent not found"</code>
<code>source_ip</code>	IP address of webhook sender	<code>54.187.174.169</code>
<code>created_at</code>	When webhook was received	<code>2023-10-15T14:29:45Z</code>

Event deduplication prevents duplicate processing when the same webhook is delivered multiple times. The challenge is determining webhook uniqueness - you cannot rely solely on the webhook ID from the provider because different event types might reuse IDs, and some providers don't include stable IDs. Instead, create a composite deduplication key combining the provider webhook ID, event type, and resource ID (like payment intent ID).

The deduplication process works by attempting to insert a webhook record with a unique constraint on the deduplication key. If the insert succeeds, this is the first time you've seen this webhook, and processing should continue. If the insert fails

due to a duplicate key violation, you've already processed this webhook, and you should return success immediately without further processing.

Key Design Insight: Webhook deduplication must happen at the database level using unique constraints, not application-level checks. Application-level checks are subject to race conditions where concurrent webhook deliveries could both pass the duplicate check before either commits their processing transaction.

Event ordering and causality present complex challenges because webhooks can arrive out of order due to network delays, provider retry logic, or load balancer routing. For example, you might receive a `charge.succeeded` webhook before the corresponding `payment_intent.processing` webhook. Your processing logic must handle these ordering issues gracefully.

One approach is to implement **causal dependency checking** where each webhook processing step verifies that prerequisite events have occurred. For instance, before processing `charge.succeeded`, verify that the payment intent exists and is in a state that allows charge completion. If prerequisites aren't met, defer processing using exponential backoff until dependencies are satisfied or timeout occurs.

Processing State	Next Action	Retry Schedule
<code>pending</code>	Attempt initial processing	Immediate
<code>retrying</code>	Retry with exponential backoff	1s, 2s, 4s, 8s, 16s, 32s
<code>failed</code>	Manual review required	None (alert operations)
<code>processed</code>	No further action	None
<code>deferred</code>	Wait for dependencies	Check every 30s for 10 minutes

The **event processing handler** contains the business logic that applies webhook events to your payment system state. This handler must be implemented as an idempotent transaction that can be safely repeated. The handler pattern typically follows these steps:

1. **Parse and validate** the webhook payload against expected schema for the event type
2. **Load current state** of the relevant payment entities (payment intent, charge, refund, etc.)
3. **Validate state transitions** to ensure the webhook event represents a legal state change
4. **Update entity state** with new information from the webhook
5. **Persist state changes** and mark webhook as processed within a single database transaction
6. **Trigger downstream actions** like sending customer notifications or updating analytics

Concurrent processing requires careful consideration because multiple webhooks for the same payment might arrive simultaneously. Use database row-level locking or optimistic concurrency control to ensure consistent state updates. Pessimistic locking (SELECT FOR UPDATE) guarantees exclusive access but can create deadlocks under high concurrency. Optimistic locking using version numbers allows higher concurrency but requires retry logic when conflicts occur.

⚠ Pitfall: Webhook Processing Outside Database Transactions Processing webhook events outside of database transactions creates race conditions where webhook processing can succeed but state updates fail, or vice versa. This

leads to webhook events being marked as processed when they actually weren't, or payment state becoming inconsistent with provider state. Always wrap webhook processing and state updates in a single atomic transaction.

Error handling and retry logic must distinguish between transient failures (database connection timeout, external service unavailable) and permanent failures (malformed webhook payload, invalid state transition). Transient failures should trigger exponential backoff retries with jitter to prevent thundering herd problems. Permanent failures should be logged, marked for manual review, and not retried automatically.

The retry mechanism should implement **circuit breaker** patterns to prevent cascading failures when downstream systems are unhealthy. If webhook processing consistently fails for a period, temporarily stop processing webhooks to allow systems to recover, while continuing to accept and store incoming webhooks for later processing.

State Reconciliation Logic

State reconciliation serves as the safety net that ensures your payment system's local state remains consistent with the authoritative state maintained by your payment provider. Think of reconciliation like balancing a checkbook - you periodically compare your records against the bank's official statement to identify and correct any discrepancies. In payment systems, these discrepancies can occur due to missed webhooks, processing failures, network partitions, or provider-side issues.

The reconciliation process operates on a fundamental assumption: **the payment provider's state is authoritative**. When discrepancies are discovered between your local records and the provider's records, your system should generally defer to the provider's version and update local state accordingly. However, this process requires careful handling to avoid overwriting legitimate local state changes or creating inconsistent audit trails.

Decision: Periodic Pull-Based Reconciliation with Event-Driven Triggers

- **Context:** Need to detect and correct payment state inconsistencies that arise from missed webhooks, processing failures, or provider-side issues
- **Options Considered:**
 1. Real-time reconciliation on every payment operation
 2. Periodic batch reconciliation with fixed schedules
 3. Event-driven reconciliation triggered by anomalies
 4. Hybrid periodic + event-driven approach
- **Decision:** Hybrid periodic + event-driven approach
- **Rationale:** Periodic reconciliation provides systematic coverage of all payments. Event-driven reconciliation enables rapid correction of detected inconsistencies. Real-time reconciliation is too expensive for every operation. Fixed schedules miss urgent corrections.
- **Consequences:** Requires maintaining reconciliation schedules and anomaly detection logic. Provides comprehensive consistency guarantees with efficient resource usage.

Reconciliation triggers determine when your system initiates state synchronization with the payment provider. Multiple trigger types work together to provide comprehensive coverage while optimizing for different scenarios and urgency levels.

Reconciliation Trigger	Frequency	Scope	Use Case
Scheduled Full Scan	Daily at 2 AM	All payments from last 30 days	Systematic consistency checking
Scheduled Incremental	Every 4 hours	Payments updated in last 24 hours	Recent payment verification
Webhook Processing Failure	Immediate	Single payment entity	Rapid error correction
Customer Service Request	On-demand	Customer's payment history	Support case resolution
Payment State Anomaly	Immediate	Related payment entities	Automated inconsistency detection
Provider Status Page Alert	Immediate	All active payments	Provider incident response

The **reconciliation algorithm** follows a systematic approach to identify, analyze, and resolve state differences. The process begins by fetching current state from both your local database and the payment provider's API for a specific set of payment entities. The comparison logic must handle the complexities of different data formats, field mappings, and temporal consistency.

State comparison logic examines multiple dimensions of payment entity state to identify meaningful differences. Simple field-by-field comparison is insufficient because providers often return additional metadata, different timestamp formats, or normalized values that don't exactly match your stored data. The comparison focuses on semantically significant differences that indicate actual state inconsistencies.

Comparison Dimension	Local Source	Provider Source	Reconciliation Action
Payment Status	<code>payment_intent.status</code>	<code>payment_intent.status</code>	Update if provider differs
Charge Amount	<code>charge.amount</code>	<code>charge.amount_captured</code>	Flag if mismatch exceeds tolerance
Refund Total	Sum of <code>refund.amount</code>	<code>charge.amount_refunded</code>	Update local refund records
Settlement Status	<code>charge.settled_at</code>	<code>charge.paid</code>	Update settlement timestamp
Failure Reason	<code>charge.failure_code</code>	<code>charge.outcome.reason</code>	Update failure details
Metadata Changes	<code>payment_intent.metadata</code>	<code>payment_intent.metadata</code>	Merge non-conflicting updates

The reconciliation process must handle **temporal consistency challenges** where state differences result from legitimate timing delays rather than actual inconsistencies. For example, a payment that shows as `processing` in your local system but `succeeded` at the provider might simply reflect normal processing delays rather than a missed webhook. The reconciliation logic should account for reasonable timing windows before treating status differences as inconsistencies.

Conflict resolution strategies define how to handle different types of state discrepancies. The strategy depends on the nature of the difference, the confidence level in local versus provider data, and the potential business impact of incorrect resolution.

Reconciliation Conflict Resolution Principles:

1. **Provider Authority:** When in doubt, the provider's state takes precedence for payment status and financial amounts
2. **Audit Preservation:** Never overwrite local records - create new records with reconciliation metadata
3. **Business Logic Validation:** Ensure state changes don't violate business rules or create inconsistent customer experience
4. **Human Review Required:** Flag complex conflicts for manual review rather than making potentially incorrect automated decisions

Batch reconciliation processing optimizes performance when reconciling large numbers of payment entities. Rather than making individual API calls for each payment, batch operations group multiple payment IDs into single provider API requests. This approach reduces API rate limit consumption and improves overall reconciliation performance.

The batch processing algorithm chunks payment entities into appropriately sized groups (typically 50-100 entities per batch), makes parallel API requests with appropriate rate limiting, and processes results in transaction batches to maintain consistency. Failed batches are retried with exponential backoff, and individual payment failures within successful batches are handled separately.

Batch Processing Parameter	Recommended Value	Rationale
Batch Size	50-100 payment entities	Balances API efficiency with memory usage
Concurrent Batches	3-5 parallel requests	Respects rate limits while improving throughput
Retry Attempts	3 attempts with exponential backoff	Handles transient API failures gracefully
Transaction Size	500 database operations	Balances transaction performance with lock duration
Progress Checkpointing	Every 1000 processed entities	Enables resumption after process interruption

Reconciliation result handling processes the outcomes of state comparison and applies appropriate corrections to local payment state. The handling logic must maintain audit trails, update related entities consistently, and trigger appropriate downstream actions like customer notifications or accounting updates.

When reconciliation identifies state differences, the correction process creates detailed audit records documenting the discrepancy, the resolution action taken, and the source of authoritative data. These audit records are essential for debugging webhook processing issues, identifying provider-side problems, and maintaining compliance with financial record-keeping requirements.

⚠ Pitfall: Reconciliation Infinite Loops Poorly designed reconciliation logic can create infinite loops where corrections made during reconciliation trigger webhook events that cause the system to revert changes, which then triggers another reconciliation cycle. Prevent this by adding reconciliation metadata to updated records and checking for recent reconciliation activity before making corrections.

Reconciliation monitoring and alerting tracks the health of the reconciliation process and identifies trends that might indicate systemic issues. Key metrics include reconciliation success rates, types and frequencies of state discrepancies, and time-to-correction for identified inconsistencies.

Reconciliation Metric	Alert Threshold	Possible Causes
Discrepancy Rate	>1% of reconciled payments	Webhook processing issues, provider problems
Failed Reconciliation Rate	>5% of reconciliation attempts	API connectivity, authentication issues
High-Value Discrepancies	Any amount >\$1000	Critical payment processing failures
Reconciliation Duration	>2 hours for daily reconciliation	Performance degradation, rate limiting
Consecutive Failures	3 failed reconciliation attempts	Persistent system or provider issues

The reconciliation system should also provide **manual reconciliation tools** for customer service teams and system operators. These tools allow on-demand reconciliation of specific payments, customer accounts, or time periods when issues are reported or suspected. The manual tools should provide detailed comparison reports showing exactly what differences were found and what corrections were applied.

Implementation Guidance

The webhook processing and reconciliation system requires careful coordination between HTTP handling, cryptographic verification, database transactions, and external API integration. This implementation guidance provides complete infrastructure components and skeleton code for the core webhook processing logic.

A. Technology Recommendations

Component	Simple Option	Advanced Option
Web Framework	Flask with Werkzeug	FastAPI with async processing
Signature Verification	<code>hmac</code> standard library	<code>cryptography</code> library with timing-safe compare
Database Operations	SQLAlchemy with PostgreSQL	SQLAlchemy async with connection pooling
Background Processing	Thread-based task queue	Celery with Redis/RabbitMQ
HTTP Client	<code>requests</code> library	<code>httpx</code> with async support and retries
Monitoring	Python <code>logging</code> module	Structured logging with correlation IDs

B. Recommended File Structure

```
payment-gateway/
  webhook/
    __init__.py
    handlers.py      ← webhook endpoint handlers
    verification.py ← signature verification logic
    processing.py   ← event processing pipeline
    reconciliation.py ← state reconciliation logic
    models.py       ← webhook data models
    exceptions.py  ← webhook-specific exceptions
  core/
    database.py     ← database session management
    payment_models.py ← payment entity models
    provider_client.py ← payment provider API client
  tests/
    test_webhook_verification.py
    test_webhook_processing.py
    test_reconciliation.py
```

C. Infrastructure Starter Code

Webhook Security Verification (webhook/verification.py)

PYTHON

```
import hmac

import hashlib

import time

from typing import Optional, Dict, Any

from dataclasses import dataclass


@dataclass

class WebhookVerificationResult:

    """Result of webhook signature verification."""

    is_valid: bool

    error_message: Optional[str] = None

    timestamp: Optional[int] = None


class WebhookVerifier:

    """Handles HMAC signature verification for incoming webhooks."""


    def __init__(self, webhook_secret: str, tolerance_seconds: int = 300):

        self.webhook_secret = webhook_secret.encode('utf-8')

        self.tolerance_seconds = tolerance_seconds


    def verify_signature(self, payload: bytes, signature_header: str) -> WebhookVerificationResult:

        """

        Verify webhook signature using HMAC-SHA256.

        Args:

            payload: Raw webhook payload bytes

            signature_header: Signature header (e.g., "t=1634567890,v1=hash...")

        Returns:

            WebhookVerificationResult with verification status and details

        """


```

```
try:

    # Parse signature header components

    sig_components = self._parse_signature_header(signature_header)

    if not sig_components:

        return WebhookVerificationResult(False, "Invalid signature header format")


    # Extract timestamp and validate age

    timestamp = sig_components.get('t')

    if not timestamp or not self._is_timestamp_valid(int(timestamp)):

        return WebhookVerificationResult(False, "Invalid or expired timestamp")


    # Verify signature using current webhook secret

    expected_signature = self._compute_signature(payload, timestamp)

    received_signature = sig_components.get('v1')


    if not received_signature or not self._secure_compare(expected_signature,
received_signature):

        return WebhookVerificationResult(False, "Signature verification failed")


    return WebhookVerificationResult(True, timestamp=int(timestamp))


except Exception as e:

    return WebhookVerificationResult(False, f"Verification error: {str(e)}")


def _parse_signature_header(self, header: str) -> Dict[str, str]:

    """Parse Stripe-style signature header into components."""

    components = {}

    for pair in header.split(', '):

        if '=' in pair:

            key, value = pair.split('=', 1)
```

```
components[key] = value

return components


def _is_timestamp_valid(self, timestamp: int) -> bool:
    """Check if webhook timestamp is within acceptable tolerance."""
    current_time = int(time.time())
    return abs(current_time - timestamp) <= self.tolerance_seconds


def _compute_signature(self, payload: bytes, timestamp: str) -> str:
    """Compute HMAC-SHA256 signature for payload and timestamp."""
    signed_payload = f"{timestamp}.".encode('utf-8') + payload
    signature = hmac.new(self.webhook_secret, signed_payload, hashlib.sha256)
    return signature.hexdigest()


def _secure_compare(self, a: str, b: str) -> bool:
    """Timing-safe string comparison to prevent timing attacks."""
    return hmac.compare_digest(a, b)
```

Webhook Models (webhook/models.py)

PYTHON

```
from sqlalchemy import Column, String, DateTime, Integer, JSON, Text, Boolean
from sqlalchemy.ext.declarative import declarative_base
from datetime import datetime
from enum import Enum

Base = declarative_base()

class WebhookProcessingStatus(Enum):
    PENDING = "pending"
    PROCESSING = "processing"
    PROCESSED = "processed"
    FAILED = "failed"
    RETRYING = "retrying"
    DEFERRED = "deferred"

class WebhookEvent(Base):
    """Webhook event record for deduplication and audit trail."""
    __tablename__ = 'webhook_events'

    id = Column(String, primary_key=True)
    event_type = Column(String, nullable=False, index=True)
    payload = Column(JSON, nullable=False)
    signature = Column(String, nullable=False)
    processed_at = Column(DateTime, nullable=True)
    processing_attempts = Column(Integer, default=0)
    processing_status = Column(String, default=WebhookProcessingStatus.PENDING.value, index=True)
    last_processing_error = Column(Text, nullable=True)
    source_ip = Column(String, nullable=True)
    created_at = Column(DateTime, default=datetime.utcnow, nullable=False)

    # Deduplication key combines provider webhook ID, event type, and resource ID
```

```
deduplication_key = Column(String, unique=True, nullable=False, index=True)

def __repr__(self):

    return f"<WebhookEvent(id={self.id}, event_type={self.event_type}, status={self.processing_status})>"

class ReconciliationRun(Base):

    """Track reconciliation execution and results."""

    __tablename__ = 'reconciliation_runs'

    id = Column(String, primary_key=True)

    run_type = Column(String, nullable=False) # 'scheduled', 'manual', 'triggered'

    started_at = Column(DateTime, default=datetime.utcnow, nullable=False)

    completed_at = Column(DateTime, nullable=True)

    processed_count = Column(Integer, default=0)

    discrepancy_count = Column(Integer, default=0)

    error_count = Column(Integer, default=0)

    status = Column(String, default='running') # 'running', 'completed', 'failed'

    error_message = Column(Text, nullable=True)

class ReconciliationDiscrepancy(Base):

    """Record of state differences found during reconciliation."""

    __tablename__ = 'reconciliation_discrepancies'

    id = Column(String, primary_key=True)

    reconciliation_run_id = Column(String, nullable=False, index=True)

    payment_intent_id = Column(String, nullable=False, index=True)

    entity_type = Column(String, nullable=False) # 'payment_intent', 'charge', 'refund'

    field_name = Column(String, nullable=False)

    local_value = Column(JSON, nullable=True)

    provider_value = Column(JSON, nullable=True)
```

```
resolution_action = Column(String, nullable=True) # 'updated_local', 'flagged_manual',  
'no_action'  
  
resolved_at = Column(DateTime, nullable=True)  
  
created_at = Column(DateTime, default=datetime.utcnow, nullable=False)
```

D. Core Logic Skeleton Code

Webhook Event Processing (webhook/processing.py)

```
from typing import Dict, Any, Optional

from sqlalchemy.orm import Session

from sqlalchemy.exc import IntegrityError

from .models import WebhookEvent, WebhookProcessingStatus

from .verification import WebhookVerifier, WebhookVerificationResult

from core.database import get_db

from core.payment_models import PaymentIntent, Charge, Refund

import logging

import json

import uuid

logger = logging.getLogger(__name__)

class WebhookProcessor:

    """Handles webhook event processing with deduplication and state updates."""

    def __init__(self, webhook_verifier: WebhookVerifier):
        self.verifier = webhook_verifier

    def process_webhook(self, payload: bytes, signature_header: str, source_ip: str) -> Dict[str, Any]:
        """
        Main webhook processing entry point.

        Returns:
            Dict with processing status and any error information
        """

        # TODO 1: Verify webhook signature using WebhookVerifier
        # TODO 2: Parse payload JSON and extract event metadata
        # TODO 3: Create deduplication key from webhook ID + event type + resource ID
        # TODO 4: Store webhook event record with deduplication check
```

```
# TODO 5: Process webhook event based on event type

# TODO 6: Update webhook status to processed on success

# Hint: Use database transaction to ensure atomic deduplication + processing

def _store_webhook_event(self, event_data: Dict[str, Any], signature: str, source_ip: str, db: Session) -> WebhookEvent:

    """
    Store webhook event with deduplication handling.

    Args:
        event_data: Parsed webhook payload
        signature: Webhook signature header
        source_ip: Source IP address of webhook request
        db: Database session

    Returns:
        WebhookEvent record (existing if duplicate, new if first occurrence)

    """

    # TODO 1: Extract webhook ID, event type, and resource ID from event_data
    # TODO 2: Generate deduplication key combining these values
    # TODO 3: Create WebhookEvent instance with all required fields
    # TODO 4: Attempt database insert with unique constraint on deduplication_key
    # TODO 5: Handle IntegrityError for duplicate webhooks (return existing record)
    # TODO 6: Return the WebhookEvent record for further processing

    # Hint: Use try/except around db.commit() to catch duplicate key violations

def _process_event_by_type(self, webhook_event: WebhookEvent, db: Session) -> None:

    """
    Route webhook events to appropriate handlers based on event type.

    """
```

Args:

```
    webhook_event: WebhookEvent record to process
```

```
    db: Database session for state updates
```

```
"""
```

```
event_type = webhook_event.event_type
```

```
payload = webhook_event.payload
```

```
# TODO 1: Parse event type and extract base object type (payment_intent, charge, refund)
```

```
# TODO 2: Route to appropriate handler method based on event type
```

```
# TODO 3: Handle unknown event types gracefully (log warning, don't fail)
```

```
# TODO 4: Update webhook processing status and attempts counter
```

```
# TODO 5: Record any processing errors in last_processing_error field
```

```
# Hint: Use event_handlers dict to map event types to handler methods
```

```
def _handle_payment_intent_event(self, event_data: Dict[str, Any], db: Session) -> None:
```

```
"""
```

```
Handle payment intent webhook events (created, succeeded, canceled, etc.).
```

Args:

```
    event_data: Webhook payload data
```

```
    db: Database session
```

```
"""
```

```
# TODO 1: Extract payment intent ID and new status from event_data
```

```
# TODO 2: Load existing PaymentIntent record from database
```

```
# TODO 3: Validate state transition is allowed (use can_transition_to method)
```

```
# TODO 4: Update PaymentIntent status and relevant fields from webhook data
```

```
# TODO 5: Handle metadata updates, timestamp updates, error information
```

```
# TODO 6: Commit transaction or raise exception on validation failure
```

```
# Hint: Always validate business rules before applying webhook updates
```

```
def _handle_charge_event(self, event_data: Dict[str, Any], db: Session) -> None:
    """
    Handle charge webhook events (succeeded, failed, captured, etc.).

    Args:
        event_data: Webhook payload data
        db: Database session
    """

    # TODO 1: Extract charge ID and updated charge data from event_data
    # TODO 2: Load existing Charge record and related PaymentIntent
    # TODO 3: Update charge status, failure codes, network transaction ID
    # TODO 4: Update settlement information if charge succeeded
    # TODO 5: Update parent PaymentIntent status if charge state affects it
    # TODO 6: Handle 3DS authentication completion if applicable

    # Hint: Charge events may affect both Charge and PaymentIntent entities

def _handle_refund_event(self, event_data: Dict[str, Any], db: Session) -> None:
    """
    Handle refund webhook events (created, succeeded, failed).

    Args:
        event_data: Webhook payload data
        db: Database session
    """

    # TODO 1: Extract refund ID and status from event_data
    # TODO 2: Load or create Refund record as needed
    # TODO 3: Update refund status, processing timestamps, failure information
    # TODO 4: Update parent Charge refund totals and status
    # TODO 5: Validate refund amount doesn't exceed available amount
    # TODO 6: Handle partial refund accounting updates
```

```
# Hint: Refunds affect the parent charge's available refund amount
```

State Reconciliation Engine (webhook/reconciliation.py)

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

from datetime import datetime, timedelta

from sqlalchemy.orm import Session

from sqlalchemy import and_, or_

from .models import ReconciliationRun, ReconciliationDiscrepancy

from core.payment_models import PaymentIntent, Charge, Refund

from core.provider_client import ProviderClient

import logging

import uuid

logger = logging.getLogger(__name__)

class PaymentReconciliationEngine:

    """Reconciles local payment state with payment provider authoritative state."""

    def __init__(self, provider_client: ProviderClient):

        self.provider_client = provider_client

    def run_scheduled_reconciliation(self, lookback_days: int = 30) -> str:

        """
        Execute scheduled reconciliation for recent payments.

        Args:
            lookback_days: How many days back to reconcile

        Returns:
            Reconciliation run ID for tracking
        """

        # TODO 1: Create ReconciliationRun record with 'scheduled' type

        # TODO 2: Query payments updated within lookback_days period

        # TODO 3: Process payments in batches to manage memory and API limits
```

```
# TODO 4: Call reconcile_payment_batch for each batch

# TODO 5: Update ReconciliationRun with final counts and completion status

# TODO 6: Return reconciliation run ID for status tracking

# Hint: Use database transaction per batch, not per individual payment


def reconcile_single_payment(self, payment_intent_id: str) -> Dict[str, Any]:


    """
    Reconcile a specific payment against provider state.

    Args:
        payment_intent_id: Payment intent to reconcile

    Returns:
        Dict with reconciliation results and any discrepancies found

    """
    # TODO 1: Create manual ReconciliationRun record for this payment

    # TODO 2: Load local PaymentIntent and related Charge/Refund records

    # TODO 3: Fetch current payment state from provider API

    # TODO 4: Compare local vs provider state across all relevant fields

    # TODO 5: Apply reconciliation updates for any discrepancies found

    # TODO 6: Return summary of changes made and discrepancies resolved

    # Hint: Handle cases where payment doesn't exist locally or at provider


    def _reconcile_payment_batch(self, payment_intent_ids: List[str], reconciliation_run_id: str, db: Session) -> Dict[str, int]:


        """
        Reconcile a batch of payments efficiently.

        Args:
            payment_intent_ids: List of payment intent IDs to reconcile
```

```

    reconciliation_run_id: Parent reconciliation run ID

    db: Database session

Returns:
    Dict with counts of processed, discrepancies, errors

"""

# TODO 1: Load local payment state for all payments in batch

# TODO 2: Fetch provider state for all payments using batch API

# TODO 3: Compare local vs provider state for each payment

# TODO 4: Generate ReconciliationDiscrepancy records for differences

# TODO 5: Apply approved reconciliation updates to local state

# TODO 6: Return processing statistics for this batch

# Hint: Use provider batch APIs to minimize API calls and rate limiting

def _compare_payment_states(self, local_payment: PaymentIntent, provider_payment: Dict[str, Any]) -> List[Dict[str, Any]]:
    """

    Compare local payment state against provider state.

Args:
    local_payment: Local PaymentIntent record with charges/refunds
    provider_payment: Provider payment data from API

Returns:
    List of discrepancies found between local and provider state

"""

discrepancies = []

# TODO 1: Compare payment intent status between local and provider
# TODO 2: Compare charge amounts, statuses, and settlement information

```

```

# TODO 3: Compare refund totals and individual refund records

# TODO 4: Compare metadata that affects business logic

# TODO 5: Check for missing charges or refunds in local state

# TODO 6: Return list of discrepancy objects with resolution recommendations

# Hint: Focus on business-critical differences, ignore cosmetic variations


def _resolve_discrepancy(self, discrepancy: Dict[str, Any], db: Session) -> bool:
    """
    Apply resolution for a detected state discrepancy.

    Args:
        discrepancy: Discrepancy data with local/provider values
        db: Database session for updates

    Returns:
        True if discrepancy was resolved, False if manual review required
    """

    # TODO 1: Determine resolution strategy based on discrepancy type

    # TODO 2: Validate that resolution won't violate business rules

    # TODO 3: Apply updates to local payment entities as needed

    # TODO 4: Create audit trail of reconciliation changes made

    # TODO 5: Handle cases requiring manual review (complex conflicts)

    # TODO 6: Return success status of resolution attempt

    # Hint: Some discrepancies need human review rather than automatic resolution

```

E. Language-Specific Hints

- **Webhook Signature Verification:** Use `hmac.compare_digest()` for timing-safe signature comparison to prevent timing attacks
- **Database Deduplication:** Use `ON CONFLICT` clauses in PostgreSQL or `IntegrityError` handling in SQLAlchemy for atomic deduplication
- **JSON Handling:** Use `json.loads()` with error handling for webhook payload parsing - malformed JSON should return 400 Bad Request

- **Async Processing:** Consider `asyncio` and `httpx` for high-volume webhook processing to handle concurrent webhook deliveries
- **Rate Limiting:** Use `time.sleep()` with exponential backoff for provider API calls during reconciliation
- **Logging:** Include correlation IDs in all log messages to trace webhook processing across components

F. Milestone Checkpoint

Verification Steps:

1. **Webhook Receipt:** `curl -X POST localhost:8000/webhooks -H "Stripe-Signature: t=1234567890,v1=invalid" -d '{"test": "data"}'` should return 401 Unauthorized
2. **Signature Verification:** Send webhook with valid HMAC signature should return 200 OK and create `WebhookEvent` record in database
3. **Deduplication:** Send same webhook twice - second request should return 200 OK but not create duplicate database records
4. **Event Processing:** Send `payment_intent.succeeded` webhook should update corresponding `PaymentIntent` status to `succeeded`
5. **Reconciliation:** Run `python -m webhook.reconciliation --payment-intent pi_test123` should compare local vs provider state and report any differences

Expected Database State:

- `webhook_events` table contains received webhooks with `processed_at` timestamps
- `payment_intents` table reflects status updates from webhook processing
- `reconciliation_runs` and `reconciliation_discrepancies` tables track reconciliation execution
- No duplicate webhook processing (verify using `deduplication_key` uniqueness)

Warning Signs:

- Webhook signature verification passing with invalid signatures (security vulnerability)
- Duplicate payment status updates from repeated webhook processing (idempotency failure)
- Reconciliation finding many discrepancies (indicates webhook processing problems)
- High webhook processing latency (database transaction or API performance issues)

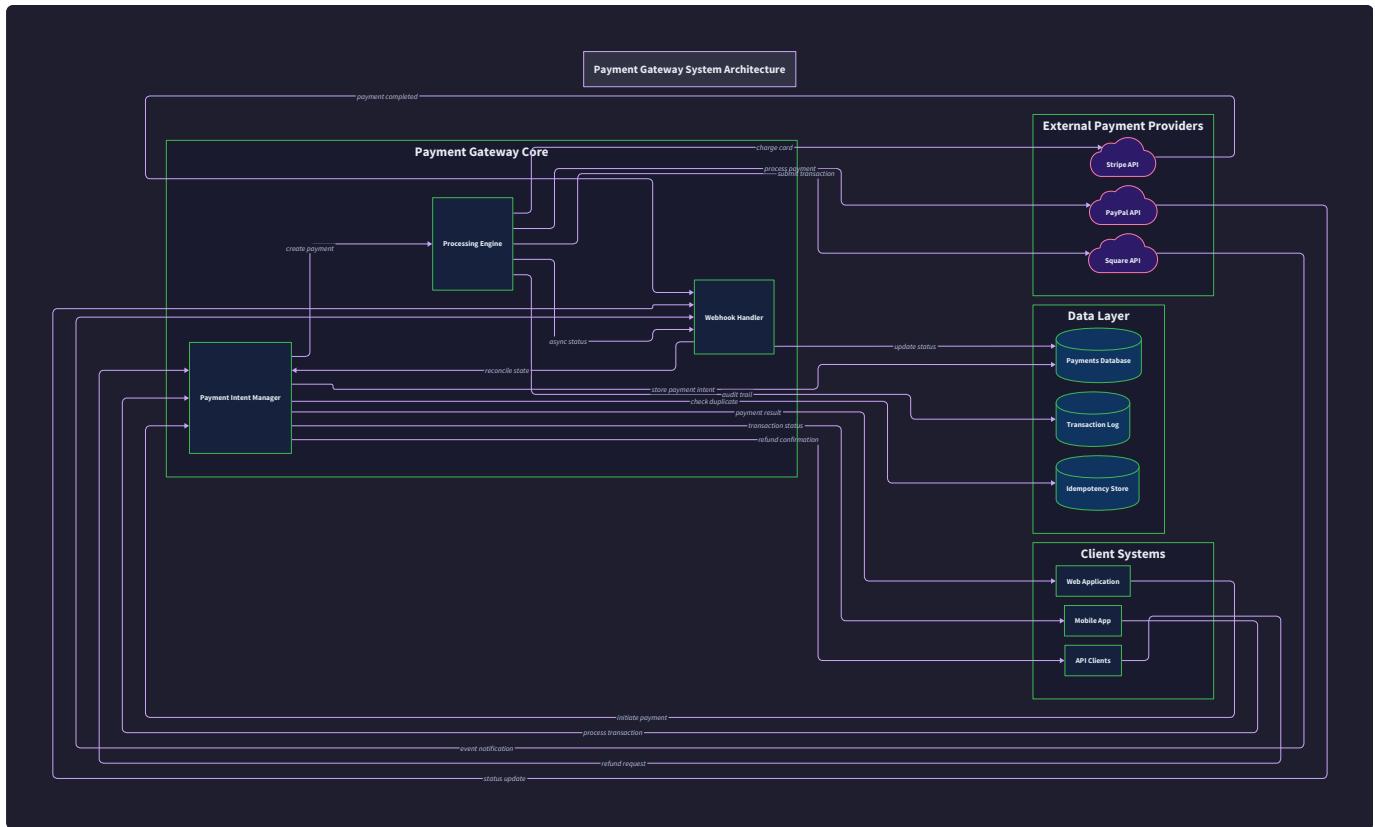
Component Interactions and Data Flow

Milestone(s): All milestones - component interaction patterns support payment intents (Milestone 1), payment processing (Milestone 2), refunds and disputes (Milestone 3), and webhook reconciliation (Milestone 4)

Think of a payment system like a complex orchestra where each musician (component) must play their part in perfect timing and harmony. Just as the conductor coordinates when the violins, trumpets, and drums each contribute to the symphony, our payment gateway coordinates when the payment intent manager, processing engine, webhook handler, and external providers each execute their responsibilities. The difference is that in payments, a missed note or wrong timing can mean lost money or compliance violations - so the coordination must be absolutely precise and recoverable.

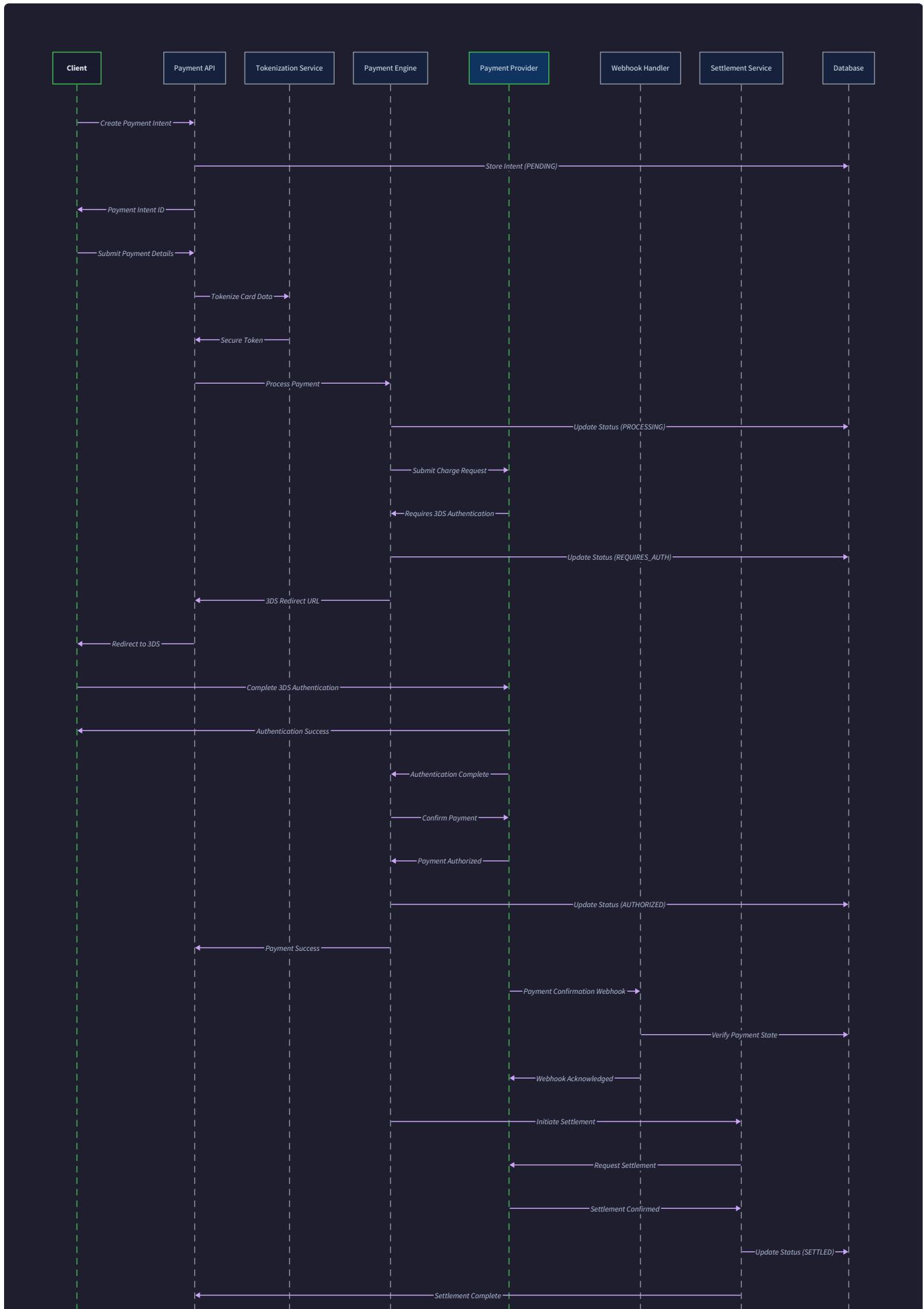
This section details how all system components communicate with each other, the exact message formats they exchange, and the precise sequence of operations for both successful and error scenarios. Understanding these interactions is critical

because payment processing involves multiple external services, asynchronous operations, and strict consistency requirements.



Complete Payment Flow Sequence

The complete payment flow represents the "happy path" - when everything works correctly from payment intent creation through final webhook confirmation. This sequence involves coordination between the client application, our payment gateway components, external payment providers, and webhook processing systems.





Phase 1: Payment Intent Creation with Idempotency

The payment flow begins when a client creates a payment intent. This phase establishes the payment commitment and handles duplicate request prevention through idempotency keys.

Step-by-Step Payment Intent Creation:

- Client Request Reception:** The payment gateway receives a `POST /payment-intents` request containing the payment amount, currency, idempotency key, and optional metadata from the client application.
- Idempotency Key Validation:** The system validates the idempotency key format and checks the `IdempotencyKey` table for existing entries with the same key. If found, it compares the request hash to ensure parameters match.
- Request Parameter Hashing:** The system creates a SHA-256 hash of essential request parameters (amount, currency, metadata) to detect parameter mismatches for the same idempotency key.
- Database Transaction Initiation:** A database transaction begins to ensure atomic creation of both the `PaymentIntent` and `IdempotencyKey` records.
- Payment Intent Record Creation:** The system creates a new `PaymentIntent` with status `created`, generates a unique client secret, and sets an expiration time (typically 24 hours).
- Idempotency Key Storage:** An `IdempotencyKey` record is created linking the key to the payment intent, storing the request hash and response details for future duplicate detection.
- Response Generation:** The system returns the payment intent ID, client secret, status, and amount to the client, which can then proceed to collect payment method details.

Payment Intent Creation Message Formats:

Field	Type	Direction	Description
<code>amount</code>	int	Request	Payment amount in cents (e.g., 1000 = \$10.00)
<code>currency</code>	str	Request	ISO 4217 currency code (e.g., "USD", "EUR")
<code>idempotency_key</code>	str	Request	Client-generated unique identifier for duplicate prevention
<code>metadata</code>	dict	Request	Optional key-value pairs for merchant reference
<code>payment_intent_id</code>	str	Response	Unique identifier for this payment intent
<code>client_secret</code>	str	Response	Secret token for client-side payment confirmation
<code>status</code>	str	Response	Current payment intent status (always "created" initially)
<code>created_at</code>	datetime	Response	Timestamp of payment intent creation
<code>expires_at</code>	datetime	Response	Expiration time for uncompleted payment intent

Phase 2: Payment Method Collection and Tokenization

After payment intent creation, the client collects payment method details and converts them to secure tokens for processing.

Payment Method Tokenization Sequence:

- Client Payment Method Collection:** The client application presents a secure payment form to collect card details, using either client-side encryption or hosted payment fields.
- Tokenization Request:** The client submits payment method details to the tokenization endpoint, which creates a secure `PaymentMethodToken` without storing raw card data.
- PCI Compliance Validation:** The tokenization service validates that card data is properly encrypted and that the request meets PCI DSS requirements for data handling.
- Token Generation:** A unique, non-reversible token is generated to represent the payment method, with the actual card details stored securely by the payment provider.
- Token Response:** The client receives the payment method token, which can be safely stored and transmitted for payment processing without PCI compliance concerns.

Tokenization Message Formats:

Field	Type	Direction	Description
<code>card_number</code>	str	Request	Encrypted or tokenized card number
<code>expiry_month</code>	int	Request	Card expiration month (1-12)
<code>expiry_year</code>	int	Request	Card expiration year (e.g., 2025)
<code>cvc</code>	str	Request	Card verification code (encrypted)
<code>cardholder_name</code>	str	Request	Name on the payment card
<code>payment_method_token</code>	str	Response	Secure token representing the payment method
<code>card_brand</code>	str	Response	Detected card brand (visa, mastercard, etc.)
<code>last_four_digits</code>	str	Response	Last four digits for display purposes
<code>fingerprint</code>	str	Response	Unique identifier for duplicate card detection

Phase 3: Payment Processing and 3D Secure Authentication

With the payment intent created and payment method tokenized, the system processes the actual payment, potentially including 3D Secure authentication.

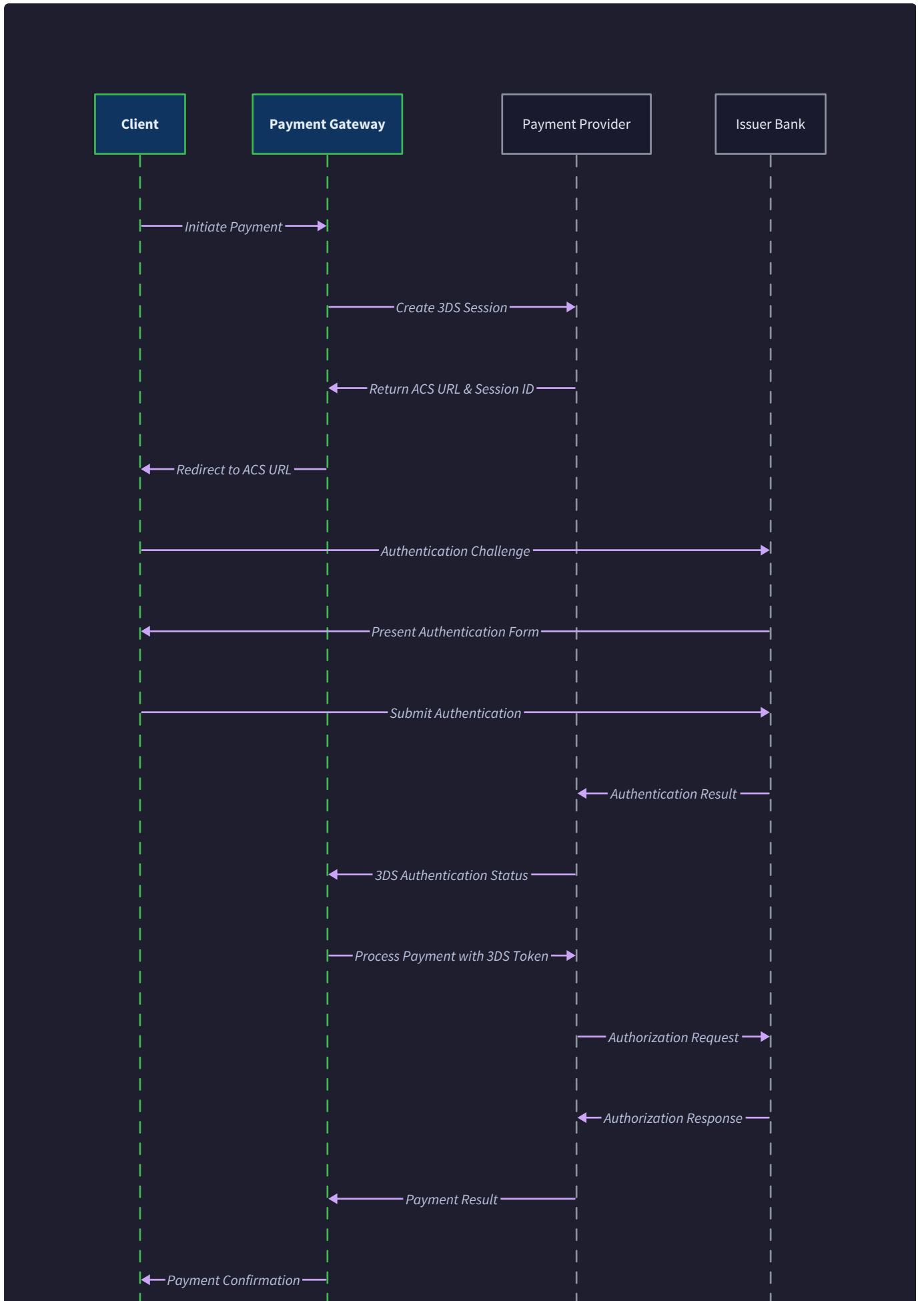
Payment Processing Sequence:

- Payment Confirmation Request:** The client submits a `POST /payment-intents/{id}/confirm` request with the payment method token and any required authentication details.
- Payment Intent Status Validation:** The system verifies the payment intent is in `created` status and has not expired, rejecting requests for intents in terminal states.

3. **Charge Creation with Provider:** The system calls `create_charge()` to submit the payment to the external payment provider, including amount, payment method token, and return URL for 3D Secure.
4. **3D Secure Challenge Detection:** If the payment provider requires Strong Customer Authentication, it returns a challenge URL and authentication token instead of immediate payment confirmation.
5. **3D Secure Redirect Handling:** For payments requiring authentication, the client redirects the user to the issuer bank's 3D Secure challenge page using the provided authentication URL.
6. **Authentication Completion:** After the user completes authentication, the issuer bank redirects back to our return URL with authentication results and tokens.
7. **Payment Confirmation:** The system calls `confirm_payment()` with the authentication token to complete the payment process and receive the final charge result.
8. **Database State Updates:** The system updates the `PaymentIntent` status and creates a `Charge` record with the provider's transaction details and authorization codes.

Payment Confirmation Message Formats:

Field	Type	Direction	Description
<code>payment_method_token</code>	str	Request	Token representing the payment method to charge
<code>return_url</code>	str	Request	URL for 3D Secure authentication redirect
<code>browser_info</code>	dict	Request	Browser details required for fraud detection
<code>payment_intent_status</code>	str	Response	Updated status (processing, requires_action, succeeded)
<code>charge_id</code>	str	Response	Unique identifier for the charge record
<code>network_transaction_id</code>	str	Response	Payment network transaction identifier
<code>authorization_code</code>	str	Response	Bank authorization code for successful payments
<code>authentication_url</code>	str	Response	3D Secure challenge URL (if authentication required)
<code>authentication_token</code>	str	Response	Token for completing 3D Secure flow



Phase 4: Webhook Processing and State Reconciliation

After payment processing, the external payment provider sends webhook notifications to confirm the final payment status and provide settlement details.

Webhook Processing Sequence:

1. **Webhook Receipt:** The payment provider sends an HTTP POST to our webhook endpoint containing event data and HMAC signature for verification.
2. **Signature Verification:** The system calls `verify_signature()` to validate the webhook payload against the HMAC signature using our shared webhook secret.
3. **Event Deduplication:** The webhook handler checks for duplicate events using the provider's event ID and our internal deduplication key to prevent double processing.
4. **Event Storage:** A `WebhookEvent` record is created with the raw payload, signature, source IP, and processing status for audit and retry purposes.
5. **Event Type Routing:** The system examines the event type and routes to the appropriate handler (`_handle_payment_intent_event`, `_handle_charge_event`, etc.).
6. **Database State Updates:** Event handlers update the corresponding `PaymentIntent`, `Charge`, or `Refund` records with the latest status information from the provider.
7. **Reconciliation Validation:** The system compares the webhook event data against our local records to detect and resolve any state discrepancies.
8. **Processing Completion:** The webhook handler returns a 200 status to acknowledge successful processing, preventing provider retry attempts.

Webhook Event Message Formats:

Field	Type	Direction	Description
event_id	str	Webhook	Unique identifier for this webhook event
event_type	str	Webhook	Type of event (payment_intent.succeeded, charge.failed, etc.)
created_timestamp	int	Webhook	Unix timestamp when the event occurred
payment_intent_id	str	Webhook	Reference to the payment intent this event concerns
charge_id	str	Webhook	Reference to the charge record (if applicable)
status	str	Webhook	Updated status for the payment entity
failure_code	str	Webhook	Standardized failure code (if payment failed)
failure_message	str	Webhook	Human-readable failure description
network_transaction_id	str	Webhook	Final network transaction identifier
settled_at	datetime	Webhook	Timestamp when funds were settled (if applicable)

Error Handling Flow Sequence

Payment processing involves numerous potential failure points, from network issues to provider errors to authentication failures. The error handling flow ensures system reliability and data consistency even when things go wrong.

Think of error handling in payments like emergency procedures in aviation - every possible failure mode has been analyzed, documented, and practiced. Pilots train extensively on what to do when engines fail, instruments malfunction, or weather turns dangerous. Similarly, our payment system must have predetermined responses to network timeouts, provider errors, authentication failures, and data inconsistencies.

Network and Connectivity Failures

Network failures are among the most common issues in payment processing, occurring when communication with external payment providers is interrupted or delayed.

Network Failure Handling Sequence:

- Timeout Detection:** When a call to the payment provider exceeds the configured timeout (typically 30 seconds), the system detects a potential network failure.
- Connection Status Classification:** The system determines whether the failure is a connection timeout (request never sent), read timeout (request sent but no response), or connection reset.
- Circuit Breaker Evaluation:** If multiple consecutive failures occur, the circuit breaker pattern prevents additional requests to protect both our system and the provider.
- Retry Strategy Implementation:** For transient failures, the system applies exponential backoff retry logic with jitter to avoid thundering herd problems.
- Payment Status Uncertainty Resolution:** When network failures occur during critical operations, the system marks the payment intent status as `processing` and initiates reconciliation.

6. **Provider Status Query:** After network recovery, the system queries the provider's API directly to determine the actual status of uncertain payments.
7. **State Correction and Notification:** Based on provider response, the system updates local records and notifies the client of the final payment outcome.

Network Failure Response Patterns:

Failure Type	Detection Method	Immediate Action	Recovery Strategy
Connection Timeout	Socket connection fails within 10s	Return temporary error to client	Retry with exponential backoff
Read Timeout	Request sent but no response in 30s	Mark payment as uncertain	Query provider status after delay
Connection Reset	TCP RST received during request	Log error and mark uncertain	Immediate retry once, then backoff
DNS Resolution	Cannot resolve provider hostname	Circuit breaker activation	Check network connectivity and DNS
SSL Handshake	TLS negotiation fails	Security alert and abort	Verify certificates and TLS configuration

Payment Provider Errors

Payment providers return various error types that require different handling strategies, from temporary rate limits to permanent card declines.

Provider Error Classification and Handling:

1. **Error Code Analysis:** The system examines the provider's error response to determine the error category (temporary, permanent, authentication, rate limit).
2. **Retriable Error Detection:** Errors like rate limits (429), temporary server errors (502, 503), and network issues are marked for retry with appropriate delays.
3. **Permanent Error Handling:** Card declines, insufficient funds, and authentication failures are immediately returned to the client without retry attempts.
4. **Rate Limit Backoff:** When providers return 429 (Too Many Requests), the system implements exponential backoff with respect for any Retry-After headers.
5. **Authentication Error Recovery:** For authentication failures with the provider, the system refreshes API credentials and retries the request once.
6. **Provider Downtime Response:** During provider outages, payments are queued for retry while clients receive graceful error messages about temporary unavailability.

Provider Error Response Handling:

Provider Response	Error Category	Client Response	Retry Strategy
card_declined	Permanent	Return decline reason	No retry - inform customer
insufficient_funds	Permanent	Return balance error	No retry - suggest different card
expired_card	Permanent	Request updated card details	No retry - collect new payment method
processing_error	Temporary	Return try again message	Exponential backoff up to 3 attempts
rate_limited	Temporary	Queue request for retry	Respect Retry-After header timing
authentication_failed	Configuration	Return system error	Refresh credentials and retry once
invalid_request	Permanent	Return validation error	No retry - fix request parameters

3D Secure Authentication Failures

3D Secure authentication introduces additional complexity with redirects, timeouts, and authentication challenges that can fail in various ways.

3D Secure Error Handling Sequence:

- Authentication Challenge Timeout:** If the user doesn't complete 3D Secure authentication within the timeout period (typically 5 minutes), the system cancels the payment attempt.
- Authentication Failure Processing:** When users fail 3D Secure challenges (wrong password, canceled authentication), the system processes the failure without retry.
- Redirect Parameter Validation:** Return URLs from 3D Secure flows are validated using `validate_return_params()` to detect tampering or corruption.
- Authentication Token Verification:** The authentication token received after 3D Secure completion is verified with the provider before completing the payment.
- Incomplete Authentication Handling:** If users abandon the 3D Secure flow without completion, the payment intent remains in `requires_action` status until expiration.

3D Secure Failure Scenarios:

Failure Type	Detection	Handling	Client Impact
User abandons 3DS	No return to callback URL	Payment intent expires after timeout	Client receives <code>payment_intent.canceled</code> webhook
Authentication fails	Provider returns auth failure	Mark charge as failed with reason	Client can retry with different payment method
Invalid return params	Parameter validation fails	Log security alert and reject	Client receives error and can restart payment
Authentication timeout	5 minute limit exceeded	Cancel payment and clean up state	Client receives timeout error
Network error during 3DS	Connection fails to issuer	Mark authentication as uncertain	Query provider for actual authentication status

Data Consistency and Reconciliation Failures

Data consistency failures occur when local payment state diverges from the provider's records, requiring reconciliation and correction.

Consistency Failure Detection and Resolution:

1. **Webhook Delivery Failure:** When webhooks fail to deliver or process correctly, local payment status may become stale or incorrect.
2. **State Divergence Detection:** Periodic reconciliation jobs compare local payment records against provider APIs to identify discrepancies.
3. **Discrepancy Classification:** Differences are classified as timing issues (acceptable delays), data errors (requiring correction), or system bugs (requiring investigation).
4. **Automated Resolution:** Simple discrepancies like status updates are automatically corrected using provider data as the authoritative source.
5. **Manual Review Triggers:** Complex discrepancies involving amounts, refunds, or disputes are flagged for manual review by operations teams.
6. **Correction Audit Trail:** All reconciliation corrections are logged with timestamps, reasons, and approval records for compliance auditing.

Data Consistency Resolution Matrix:

Discrepancy Type	Automatic Resolution	Manual Review Required	Alert Priority
Status behind provider	Update local status	No	Low
Amount mismatch	No - flag for review	Yes - financial impact	High
Missing charge record	Create from provider data	Yes - data integrity issue	Medium
Extra refund record	No - potential duplicate	Yes - money movement	High
Settlement date difference	Update to provider value	No - timing issue	Low
Dispute status mismatch	Update local status	Yes - review deadline impact	Medium

Key Design Insight: Error handling in payments requires distinguishing between recoverable and non-recoverable failures. Recoverable failures (network timeouts, rate limits) should trigger retry logic with proper backoff. Non-recoverable failures (card declines, authentication failures) should fail fast and provide clear feedback to help users resolve the issue.

API Contracts and Message Formats

API contracts define the exact structure of messages exchanged between system components, external providers, and client applications. These contracts must be precisely defined because payment systems require strict data validation and error handling.

Think of API contracts in payments like international banking protocols - every field, format, and response code must be standardized and understood by all parties. Just as SWIFT messages for international wire transfers follow exact

formatting rules that banks worldwide understand, our payment APIs must follow consistent patterns that clients, providers, and internal components can rely on.

Internal Component API Contracts

Internal APIs facilitate communication between payment gateway components, ensuring consistent data flow and proper error propagation.

Payment Intent Manager API:

Method	Parameters	Returns	Description
<code>create_payment_intent()</code>	<code>amount: int, currency: str, idempotency_key: str, metadata: dict</code>	<code>PaymentIntent</code>	Creates new payment intent with idempotency checking
<code>get_payment_intent()</code>	<code>payment_intent_id: str</code>	<code>PaymentIntent</code>	Retrieves payment intent by ID
<code>update_payment_intent_status()</code>	<code>payment_intent_id: str, new_status: PaymentIntentStatus, error_info: dict</code>	<code>bool</code>	Updates payment intent status with validation
<code>expire_payment_intents()</code>	<code>cutoff_time: datetime</code>	<code>List[str]</code>	Cancels expired payment intents, returns canceled IDs
<code>validate_payment_intent_transition()</code>	<code>current_status: PaymentIntentStatus, new_status: PaymentIntentStatus</code>	<code>bool</code>	Validates status transition is allowed

Payment Processing Engine API:

Method	Parameters	Returns	Description
<code>process_payment_intent()</code>	<code>payment_intent_id: str,</code> <code>payment_method_token:</code> <code>PaymentMethodToken, db: Session</code>	<code>Charge</code>	Main entry point for payment processing
<code>handle_three_ds_return()</code>	<code>query_params: dict, db: Session</code>	<code>Charge</code>	Processes 3D Secure authentication return
<code>create_charge()</code>	<code>payment_method_token:</code> <code>PaymentMethodToken, amount: int,</code> <code>currency: str, return_url: str,</code> <code>metadata: dict</code>	<code>ChargeResponse</code>	Submits payment to external provider
<code>confirm_payment()</code>	<code>provider_charge_id: str,</code> <code>authentication_token: str</code>	<code>ChargeResponse</code>	Completes 3D Secure authentication flow
<code>generate_return_url()</code>	<code>payment_intent_id: str,</code> <code>session_id: str</code>	<code>str</code>	Creates secure 3D Secure return URL

Refund and Dispute API:

Method	Parameters	Returns	Description
<code>create_refund()</code>	<code>charge_id: str, amount: int,</code> <code>reason: str,</code> <code>idempotency_key: str</code>	<code>Refund</code>	Creates full or partial refund
<code>get_refund_status()</code>	<code>refund_id: str</code>	<code>Refund</code>	Retrieves current refund status
<code>handle_chargeback_notification()</code>	<code>chargeback_data: dict,</code> <code>webhook_signature: str</code>	<code>bool</code>	Processes chargeback webhook
<code>submit_dispute_evidence()</code>	<code>dispute_id: str, evidence: dict,</code> <code>deadline: datetime</code>	<code>bool</code>	Submits evidence for dispute
<code>validate_refund_amount()</code>	<code>charge_id: str,</code> <code>refund_amount: int</code>	<code>bool</code>	Validates refund amount against charge and existing refunds

External Provider API Integration

External provider APIs handle actual payment processing, requiring careful request formatting and response handling.

Decision: Provider API Abstraction Strategy

- **Context:** Different payment providers (Stripe, PayPal, Adyen) have varying API structures and response formats
- **Options Considered:**
 1. Direct provider integration with provider-specific code
 2. Abstract provider interface with concrete implementations
 3. Universal payment protocol translation layer
- **Decision:** Abstract provider interface with concrete implementations
- **Rationale:** Enables multi-provider support while maintaining type safety and clear error handling patterns
- **Consequences:** Requires mapping provider-specific responses to common formats, but enables provider switching and redundancy

Payment Provider Interface:

Method	Request Format	Response Format	Description
<code>create_charge()</code>	<pre>{"amount": int, "currency": str, "payment_method": str, "return_url": str, "metadata": dict}</pre>	<pre>{"id": str, "status": str, "client_secret": str, "next_action": dict}</pre>	Initiates payment with provider
<code>retrieve_charge()</code>	<pre>{"charge_id": str}</pre>	<pre>{"id": str, "status": str, "amount": int, "failure_code": str, "network_txn_id": str}</pre>	Gets current charge status
<code>confirm_payment()</code>	<pre>{"charge_id": str, "payment_method": str, "return_url": str}</pre>	<pre>{"id": str, "status": str, "authorization_code": str}</pre>	Confirms payment after 3DS
<code>create_refund()</code>	<pre>{"charge_id": str, "amount": int, "reason": str}</pre>	<pre>{"id": str, "status": str, "refund_amount": int}</pre>	Creates refund against charge
<code>list_events()</code>	<pre>{"created_after": datetime, "limit": int, "starting_after": str}</pre>	<pre>{"events": [event], "has_more": bool}</pre>	Lists recent events for reconciliation

Provider Response Mapping:

Provider Field	Standard Field	Data Type	Transformation
<code>id</code>	<code>provider_charge_id</code>	str	Direct mapping
<code>status</code>	<code>charge_status</code>	ChargeStatus	Map to enum values
<code>amount</code>	<code>amount</code>	int	Ensure cents conversion
<code>currency</code>	<code>currency</code>	str	Validate against supported currencies
<code>failure_code</code>	<code>failure_code</code>	str	Map to standard failure codes
<code>failure_message</code>	<code>failure_message</code>	str	Direct mapping with length limits
<code>created</code>	<code>created_at</code>	datetime	Convert from Unix timestamp
<code>source.last4</code>	<code>last_four_digits</code>	str	Extract from nested payment method

Client-Facing REST API Contracts

Client-facing APIs provide the interface for merchant applications to integrate payment processing into their checkout flows.

Payment Intent Endpoints:

```
POST /v1/payment-intents
Content-Type: application/json
HTTP
Idempotency-Key: unique_key_12345

{
  "amount": 1000,
  "currency": "USD",
  "metadata": {
    "order_id": "order_12345",
    "customer_email": "user@example.com"
  }
}
```

Response Format:

```
{  
  "id": "pi_1234567890",  
  "client_secret": "pi_1234567890_secret_abcdef",  
  "amount": 1000,  
  "currency": "USD",  
  "status": "created",  
  "created_at": "2023-10-15T14:30:00Z",  
  "expires_at": "2023-10-16T14:30:00Z",  
  "metadata": {  
    "order_id": "order_12345",  
    "customer_email": "user@example.com"  
  }  
}
```

JSON

Payment Confirmation Endpoint:

```
POST /v1/payment-intents/{id}/confirm  
Content-Type: application/json  
  
{  
  "payment_method_token": "pm_card_visa_4242",  
  "return_url": "https://merchant.com/payment-return",  
  "browser_info": {  
    "user_agent": "Mozilla/5.0...",  
    "accept_header": "text/html,application/xhtml+xml",  
    "language": "en-US",  
    "color_depth": 24,  
    "screen_height": 1080,  
    "screen_width": 1920,  
    "timezone": -300  
  }  
}
```

HTTP

Webhook Event Message Contracts

Webhook events provide asynchronous notifications about payment status changes, requiring standardized message formats for reliable processing.

Webhook Event Structure:

Field	Type	Required	Description
<code>id</code>	str	Yes	Unique event identifier
<code>type</code>	str	Yes	Event type (payment_intent.succeeded, charge.failed)
<code>created</code>	int	Yes	Unix timestamp when event occurred
<code>data</code>	object	Yes	Event-specific payload data
<code>request</code>	object	No	Details about the API request that triggered the event
<code>pending_webhooks</code>	int	Yes	Number of pending webhook deliveries

Payment Intent Webhook Events:

Event Type	Trigger	Data Payload Fields
<code>payment_intent.created</code>	New payment intent created	<code>payment_intent: PaymentIntent</code>
<code>payment_intent.requires_action</code>	3D Secure authentication needed	<code>payment_intent: PaymentIntent, next_action: dict</code>
<code>payment_intent.processing</code>	Payment submitted to provider	<code>payment_intent: PaymentIntent, charge: Charge</code>
<code>payment_intent.succeeded</code>	Payment completed successfully	<code>payment_intent: PaymentIntent, charge: Charge</code>
<code>payment_intent.payment_failed</code>	Payment failed permanently	<code>payment_intent: PaymentIntent, last_payment_error: dict</code>
<code>payment_intent.canceled</code>	Intent expired or manually canceled	<code>payment_intent: PaymentIntent, cancellation_reason: str</code>

Error Response Standardization:

All API endpoints return consistent error response formats to enable proper error handling by client applications.

Standard Error Response:

JSON

```
{  
  "error": {  
    "type": "card_error",  
    "code": "card_declined",  
    "message": "Your card was declined.",  
    "decline_code": "insufficient_funds",  
    "payment_intent": {  
      "id": "pi_1234567890",  
      "status": "requires_payment_method"  
    }  
  }  
}
```

Error Type Classification:

Error Type	HTTP Status	Description	Client Action
api_error	500	Internal server error	Retry after delay
authentication_error	401	Invalid API credentials	Check API keys
card_error	402	Payment method declined	Request different payment method
idempotency_error	400	Idempotency key reused with different params	Use new idempotency key
invalid_request_error	400	Invalid parameters	Fix request parameters
rate_limit_error	429	Too many requests	Implement request throttling

Critical Implementation Note: All monetary amounts in API contracts MUST be represented as integers in the smallest currency unit (cents for USD, pence for GBP) to avoid floating-point precision errors. A \$10.00 payment is represented as `"amount": 1000`, never as `"amount": 10.0`.

Implementation Guidance

The component interaction patterns require careful coordination between multiple services, proper error handling, and reliable message processing. This guidance provides practical implementation approaches for building these interaction patterns.

Technology Recommendations:

Component	Simple Option	Advanced Option
HTTP Client	<code>requests</code> library with retry logic	<code>aiohttp</code> for async processing
Message Serialization	<code>json</code> standard library	<code>pydantic</code> for validation
Webhook Verification	<code>hmac</code> standard library	<code>cryptography</code> library
Database Sessions	<code>SQLAlchemy</code> with context managers	Connection pooling with <code>asyncpg</code>
API Documentation	Manual OpenAPI spec	<code>FastAPI</code> auto-generated docs
Request Validation	Manual parameter checking	<code>marshmallow</code> schemas

Recommended Project Structure:

```

payment-gateway/
├── src/
│   ├── api/
│   │   ├── __init__.py
│   │   ├── payment_intents.py      ← Client-facing REST endpoints
│   │   ├── webhooks.py            ← Webhook processing endpoints
│   │   └── middleware.py         ← Request logging and validation
│   ├── services/
│   │   ├── __init__.py
│   │   ├── payment_processor.py  ← Core payment processing logic
│   │   ├── provider_client.py   ← External provider integration
│   │   └── reconciliation.py    ← State reconciliation service
│   ├── models/
│   │   ├── __init__.py
│   │   ├── payment_intent.py     ← Data model definitions
│   │   └── webhook_event.py     ← Webhook event models
│   ├── utils/
│   │   ├── __init__.py
│   │   ├── crypto.py             ← Signature verification utilities
│   │   └── retry.py              ← Retry logic helpers
│   └── main.py                 ← Application entry point
└── tests/
    ├── test_api.py             ← API endpoint tests
    ├── test_services.py        ← Service layer tests
    └── test_integration.py     ← End-to-end tests
└── requirements.txt

```

HTTP Client Infrastructure (Complete Implementation):

```
# utils/http_client.py

import time
import logging
import requests

from typing import Optional, Dict, Any
from dataclasses import dataclass
from enum import Enum

class RetryStrategy(Enum):
    EXPONENTIAL_BACKOFF = "exponential"
    FIXED_DELAY = "fixed"
    NO_RETRY = "none"

    @dataclass
    class RetryConfig:
        max_attempts: int = 3
        initial_delay: float = 1.0
        max_delay: float = 60.0
        backoff_multiplier: float = 2.0
        strategy: RetryStrategy = RetryStrategy.EXPOENTIAL_BACKOFF

    class PaymentProviderClient:
        def __init__(self, base_url: str, api_key: str, retry_config: RetryConfig):
            self.base_url = base_url
            self.api_key = api_key
            self.retry_config = retry_config
            self.session = requests.Session()
            self.session.headers.update({
                'Authorization': f'Bearer {api_key}',
                'Content-Type': 'application/json',
                'User-Agent': 'PaymentGateway/1.0'
            })
```

```
)
```



```
def make_request(self, method: str, endpoint: str, data: Optional[Dict[str, Any]] = None) ->
Dict[str, Any]:
```

```
    url = f"{self.base_url}/{endpoint.lstrip('/')}"
```



```
    for attempt in range(self.retry_config.max_attempts):
```

```
        try:
```

```
            response = self.session.request(method, url, json=data, timeout=30)
```



```
            if response.status_code == 429: # Rate limited
```

```
                retry_after = int(response.headers.get('Retry-After', '60'))
```

```
                if attempt < self.retry_config.max_attempts - 1:
```

```
                    time.sleep(retry_after)
```

```
                    continue
```



```
            response.raise_for_status()
```

```
            return response.json()
```



```
    except (requests.exceptions.ConnectTimeout,
```

```
            requests.exceptions.ReadTimeout,
```

```
            requests.exceptions.ConnectionError) as e:
```



```
        if attempt < self.retry_config.max_attempts - 1:
```

```
            delay = self._calculate_retry_delay(attempt)
```

```
            logging.warning(f"Request failed (attempt {attempt + 1}), retrying in {delay}s: {e}")
```

```
            time.sleep(delay)
```

```
            continue
```

```
        else:
```

```

        logging.error(f"Request failed after {self.retry_config.max_attempts} attempts:
{e}")

        raise

    except requests.exceptions.HTTPError as e:

        # Don't retry 4xx errors (client errors)

        if 400 <= e.response.status_code < 500:

            raise

        # Retry 5xx errors (server errors)

        if attempt < self.retry_config.max_attempts - 1:

            delay = self._calculate_retry_delay(attempt)

            time.sleep(delay)

            continue

        else:

            raise

def _calculate_retry_delay(self, attempt: int) -> float:

    if self.retry_config.strategy == RetryStrategy.NO_RETRY:

        return 0

    elif self.retry_config.strategy == RetryStrategy.FIXED_DELAY:

        return self.retry_config.initial_delay

    else: # EXPONENTIAL_BACKOFF

        delay = self.retry_config.initial_delay * (self.retry_config.backoff_multiplier ** attempt)

        return min(delay, self.retry_config.max_delay)

```

Webhook Signature Verification (Complete Implementation):

```
# utils/webhook_security.py

import hmac

import hashlib

import time

from typing import Dict, Any, Optional


class WebhookVerificationError(Exception):

    pass


class WebhookSignatureVerifier:

    def __init__(self, webhook_secret: str, tolerance_seconds: int = 300):

        self.webhook_secret = webhook_secret.encode('utf-8')

        self.tolerance_seconds = tolerance_seconds


    def verify_signature(self, payload: str, signature_header: str, timestamp: Optional[str] = None) -> bool:

        try:

            # Parse signature header (format: "t=timestamp,v1=signature")

            sig_parts = {}

            for part in signature_header.split(','):

                key, value = part.split('=', 1)

                sig_parts[key] = value


            if 't' not in sig_parts or 'v1' not in sig_parts:

                raise WebhookVerificationError("Invalid signature header format")


            # Verify timestamp to prevent replay attacks

            webhook_timestamp = int(sig_parts['t'])

            current_timestamp = int(time.time())


            if abs(current_timestamp - webhook_timestamp) > self.tolerance_seconds:
```

```
        raise WebhookVerificationError("Webhook timestamp outside tolerance window")

# Construct signed payload: timestamp.payload

signed_payload = f"{webhook_timestamp}.{payload}"

# Calculate expected signature

expected_signature = hmac.new(
    self.webhook_secret,
    signed_payload.encode('utf-8'),
    hashlib.sha256
).hexdigest()

# Compare signatures using constant-time comparison

received_signature = sig_parts['v1']

return hmac.compare_digest(expected_signature, received_signature)

except (ValueError, KeyError) as e:

    raise WebhookVerificationError(f"Signature verification failed: {e}")
```

Core Component Interaction Logic (Skeleton with TODOs):

```
# services/payment_flow_coordinator.py

from typing import Dict, Any, Optional

from sqlalchemy.orm import Session

from models.payment_intent import PaymentIntent, PaymentIntentStatus

from models.charge import Charge, ChargeStatus

from utils.http_client import PaymentProviderClient


class PaymentFlowCoordinator:

    def __init__(self, provider_client: PaymentProviderClient):

        self.provider_client = provider_client


    def execute_complete_payment_flow(self,
                                      payment_intent_id: str,
                                      payment_method_token: str,
                                      db: Session) -> Dict[str, Any]:
        """
        Orchestrates the complete payment flow from intent confirmation to webhook processing.

        Returns the final payment result with all status updates.
        """

        # TODO 1: Load and validate payment intent from database
        # - Query PaymentIntent by ID
        # - Verify status is 'created' and not expired
        # - Raise appropriate errors for invalid states

        # TODO 2: Update payment intent status to 'processing'
        # - Use database transaction for atomic update
        # - Record processing start timestamp
        # - Handle concurrent update conflicts

        # TODO 3: Create charge with payment provider
```

```

# - Call self.provider_client.make_request() with charge data

# - Include payment_method_token, amount, currency, return_url

# - Handle provider-specific response format


# TODO 4: Process provider response for 3D Secure requirements

# - Check if response includes 'next_action' for authentication

# - Update payment intent status to 'requires_action' if needed

# - Generate secure return URL for 3DS flow


# TODO 5: Create local Charge record with provider response data

# - Map provider fields to local Charge model

# - Store network_transaction_id and authorization_code

# - Handle partial data from pending 3DS authentication


# TODO 6: Return appropriate response based on payment status

# - For immediate success: return charge details

# - For 3DS required: return authentication URL and token

# - For failures: return error details and suggested actions


pass


def handle_webhook_state_update(self, webhook_event: Dict[str, Any], db: Session) -> bool:
    """
    Processes webhook events to update local payment state.

    Returns True if processing was successful.

    """

    # TODO 1: Extract event type and payment entity ID from webhook data

    # - Parse event_type field (payment_intent.succeeded, charge.failed, etc.)

    # - Extract payment_intent_id or charge_id from nested data

    # - Validate required fields are present

```

```
# TODO 2: Load corresponding local record from database

# - Query appropriate model (PaymentIntent, Charge, Refund)

# - Handle cases where local record doesn't exist yet

# - Lock record for update to prevent race conditions


# TODO 3: Validate webhook event against current local state

# - Check if event timestamp is newer than last update

# - Verify state transition is valid (use can_transition_to())

# - Detect and handle out-of-order event delivery


# TODO 4: Apply webhook data to local record

# - Update status, timestamps, and provider-specific fields

# - Preserve local metadata and tracking information

# - Handle partial updates vs complete record replacement


# TODO 5: Trigger any downstream processing required

# - Send notifications for terminal status changes

# - Update related records (refunds, disputes)

# - Enqueue reconciliation checks if needed


pass
```

Milestone Checkpoint 1: Basic Component Communication After implementing the HTTP client and basic coordination logic:

```
# Test the HTTP client retry logic                                BASH

python -m pytest tests/test_http_client.py -v

# Expected output:

# test_exponential_backoff_retry PASSED

# test_rate_limit_handling PASSED

# test_permanent_error_no_retry PASSED
```

Manual verification: Start a mock payment provider server that returns different HTTP status codes. Verify that the client properly retries 5xx errors but not 4xx errors.

Milestone Checkpoint 2: End-to-End Payment Flow After implementing the complete payment coordination:

```
# Run integration tests                                         BASH

python -m pytest tests/test_payment_flow.py -v

# Expected output:

# test_successful_payment_flow PASSED

# test_3ds_authentication_flow PASSED

# test_payment_failure_handling PASSED
```

Manual verification: Create a payment intent, submit a test card that requires 3D Secure, complete the authentication flow, and verify webhook processing updates the final status.

Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
Payment stuck in "processing"	Webhook delivery failed	Check webhook endpoint logs and provider dashboard	Implement webhook retry mechanism and manual reconciliation
Duplicate charge creation	Race condition in payment processing	Check for concurrent requests with same idempotency key	Add database-level unique constraints on idempotency keys
3D Secure redirect fails	Invalid return URL or session timeout	Verify return URL generation and session management	Implement proper session validation and error handling
Provider API calls timing out	Network latency or provider issues	Monitor response times and error rates	Increase timeout values and implement circuit breaker
State inconsistency between components	Missing database transaction boundaries	Review database transaction scope and rollback handling	Wrap multi-step operations in proper database transactions

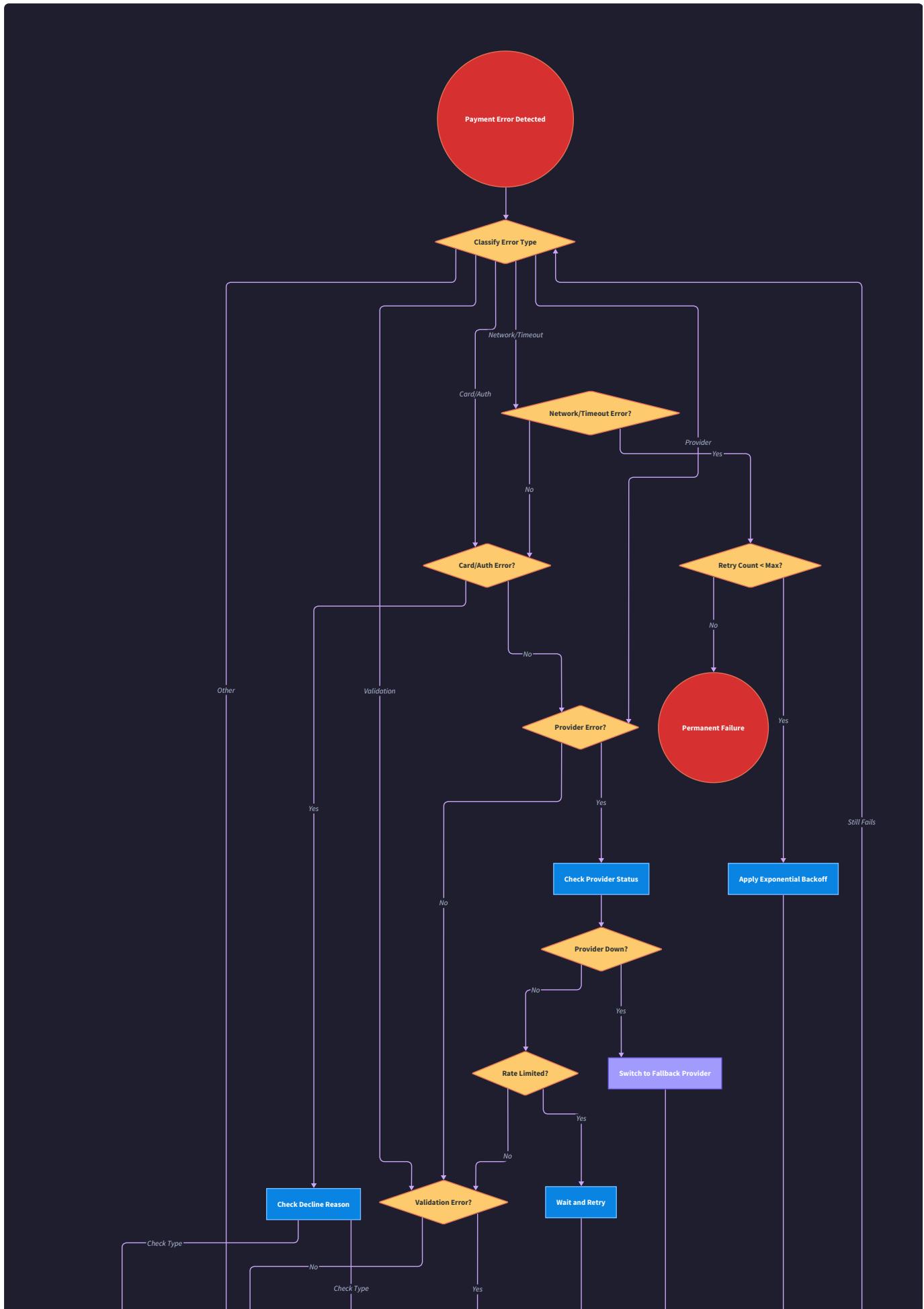
Error Handling and Edge Cases

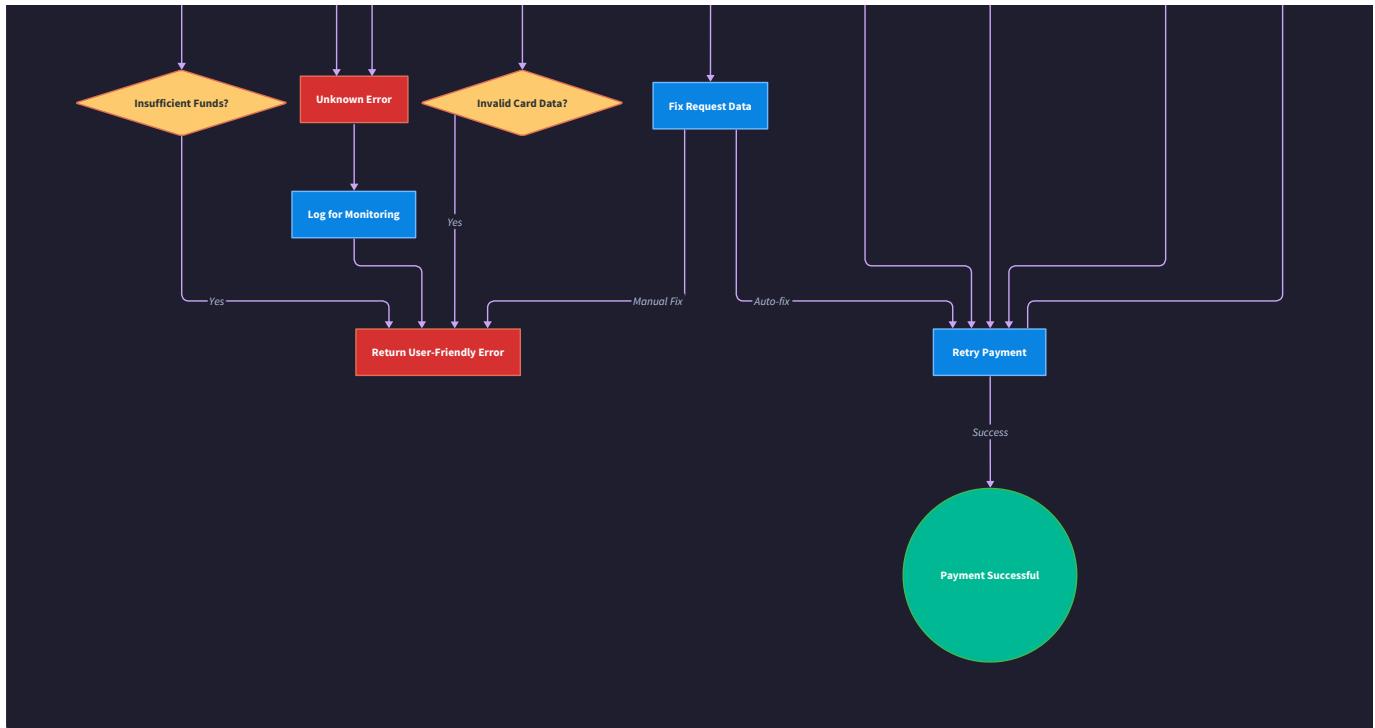
Milestone(s): All milestones - error handling spans payment intents (Milestone 1), payment processing (Milestone 2), refunds and disputes (Milestone 3), and webhook reconciliation (Milestone 4)

Building a payment gateway is like constructing a financial fortress - it must withstand every conceivable attack, failure, and edge case while protecting the money flowing through it. Unlike typical web applications where a failed request can simply return an error, payment systems deal with real money where failures have financial consequences. A network timeout during payment processing doesn't just mean "try again later" - it creates uncertainty about whether money actually moved, requiring sophisticated reconciliation to resolve.

The mental model for payment error handling resembles air traffic control systems: multiple redundant safety mechanisms, clear protocols for every failure scenario, and the ability to maintain safe operations even when individual components fail. Just as air traffic controllers have backup communication systems and predetermined procedures for radio failures, payment systems need comprehensive error handling strategies that account for partial failures, network partitions, and provider outages.

This section establishes the error handling foundation that makes payment processing reliable across all system components. Every payment operation - from creating intents with idempotency keys to processing webhooks - depends on these error handling patterns to maintain data consistency and financial accuracy.





Failure Mode Analysis

Payment systems face a complex landscape of potential failures, each requiring specific detection and recovery strategies. Understanding these failure modes is crucial because payment operations are inherently distributed - they involve coordination between your application, external payment providers, issuing banks, and card networks. Each integration point introduces new failure possibilities.

The failure taxonomy for payment systems differs fundamentally from typical web applications. While a failed web request might inconvenience a user, a failed payment operation can result in money being charged without goods delivered, refunds being processed twice, or chargebacks going uncontested. This makes comprehensive failure analysis not just good engineering practice, but a business necessity.

Network and Communication Failures

Network failures represent the most common category of payment processing issues. These failures are particularly challenging because they create uncertainty - when a network request times out during payment processing, the system cannot determine whether the payment succeeded at the provider level.

Failure Mode	Detection Method	Business Impact	Recovery Strategy
Provider API timeout during charge creation	HTTP request timeout (30-60 seconds)	Payment may have succeeded but status unknown	Query provider API with original idempotency key
Provider API timeout during 3DS confirmation	Webhook timeout or status polling	Customer completed authentication but charge uncertain	Reconcile charge status via provider API
Network partition during webhook delivery	Missing webhook events for known payments	Local payment state becomes stale	Run immediate reconciliation scan
DNS resolution failure for provider API	Connection establishment error	All new payments blocked	Circuit breaker pattern with cached IP fallback
SSL/TLS handshake failure	Certificate validation error	Payment processing completely blocked	Retry with exponential backoff, alert operations team
Intermittent 5xx errors from provider	HTTP 502, 503, 504 status codes	Individual payments failing randomly	Exponential backoff with jitter, track error rates

Network failures require careful timeout configuration. Provider API calls should use timeouts shorter than your own API response requirements - if your payment API promises 30-second response times, provider calls must timeout in 20-25 seconds to allow time for error handling and response formatting.

Critical Insight: Network timeouts during payment operations create the "money in flight" problem - you cannot assume the payment failed just because you didn't receive a response. Always implement status reconciliation for timeout scenarios.

Provider-Specific Failures

External payment providers introduce their own failure modes based on their internal systems, rate limiting, and business rules. These failures often carry specific error codes that require different handling strategies.

Provider Error Type	Common Error Codes	Retry Strategy	Required Action
Rate limiting	<code>rate_limit_exceeded</code>	Exponential backoff with provider-specific limits	Honor retry-after headers, implement token bucket
Insufficient funds	<code>insufficient_funds</code> , <code>declined</code>	No retry - terminal failure	Mark charge as failed, update payment intent status
Invalid payment method	<code>invalid_card_number</code> , <code>expired_card</code>	No retry - user error	Return validation error to client
Provider maintenance	<code>temporarily_unavailable</code>	Exponential backoff up to 15 minutes	Circuit breaker, consider backup provider
Authentication errors	<code>invalid_api_key</code> , <code>permission_denied</code>	No retry - configuration issue	Alert operations, check credential rotation
Idempotency key conflicts	<code>idempotency_key_reused</code>	Check if previous request succeeded	Query existing payment status, return if successful

Provider failures require understanding the semantics of different error codes. Some errors indicate permanent failures that should never be retried (invalid card numbers), while others suggest temporary issues that may resolve with time (rate limiting).

Decision: Provider Error Code Mapping

- **Context:** Different providers use different error code formats and semantics, making unified error handling difficult
- **Options Considered:**
 1. Pass through provider error codes directly
 2. Map all provider errors to internal error taxonomy
 3. Hybrid approach with provider passthrough plus internal classification
- **Decision:** Map provider errors to internal taxonomy with original code preserved
- **Rationale:** Internal taxonomy enables consistent retry logic and client error handling while preserving provider-specific details for debugging
- **Consequences:** Requires maintaining mapping tables for each provider but enables provider-agnostic error handling logic

Data Consistency Failures

Payment systems must maintain strict data consistency across multiple operations - creating payment intents, processing charges, handling webhooks, and reconciling state. Failures can leave the system in partially updated states that compromise financial accuracy.

Consistency Issue	Symptoms	Detection Method	Resolution Strategy
Payment intent created but charge processing failed	Intent exists in <code>created</code> status indefinitely	Check for old intents without corresponding charges	Cancel expired intents, refund any orphaned charges
Charge succeeded at provider but webhook missed	Local charge stuck in <code>pending</code> , provider shows <code>succeeded</code>	Reconciliation discovers status mismatch	Update local charge status, trigger downstream notifications
Webhook processed but database update failed	Webhook marked processed, payment status not updated	Transaction rollback detection	Retry webhook processing with idempotency protection
Partial refund amounts don't sum correctly	Total refunds exceed original charge amount	Validation during refund creation	Reject refund, audit existing refund records
Duplicate webhook processing	Same event processed multiple times	Multiple database records for single event	Implement webhook deduplication using event IDs
3DS authentication completed but charge confirmation failed	Customer completed authentication, charge not finalized	Status polling after authentication timeout	Query provider for charge status, manual reconciliation if needed

Data consistency issues require careful transaction boundary design. Related database updates should occur within single transactions where possible, with compensation logic for operations that cannot be atomic.

State Machine Violations

Payment entities follow strict state machines - payment intents, charges, refunds, and disputes each have defined valid transitions. Violations of these state machines indicate bugs or external manipulation.

State Violation	Valid Prevention	Error Handling
Payment intent transitions from <code>succeeded</code> to <code>processing</code>	Check <code>can_transition_to()</code> before updates	Log error, reject transition, alert monitoring
Charge moves from <code>failed</code> to <code>succeeded</code>	Database constraints on status transitions	Treat as provider data correction, audit carefully
Refund created against non-existent charge	Foreign key constraints	Return 404 error, log potential fraud attempt
Dispute created for already-refunded payment	Business logic validation	May be valid - partial refunds can still be disputed
Multiple charges against single payment intent	Unique constraint on <code>payment_intent_id + charge status</code>	Second charge should reuse existing successful charge
Webhook updates payment to invalid state	Webhook validation before processing	Quarantine webhook, manual review required

State machine violations often indicate either bugs in business logic or attempts at system manipulation. All violations should be logged with high severity and trigger alerting.

Retry and Backoff Strategies

Retry strategies for payment systems require balancing responsiveness against system stability. Unlike simple web requests, payment operations have time-sensitive characteristics - customers expect quick responses, authentication flows have timeouts, and providers may interpret repeated requests as potential fraud.

The mental model for payment retry strategies resembles emergency response protocols: immediate action for life-threatening situations, measured escalation for serious problems, and careful evaluation before dramatic interventions. Payment retries must be similarly calibrated - aggressive retries for transient network issues but conservative approaches for provider errors that might indicate deeper problems.

Exponential Backoff Implementation

Exponential backoff with jitter provides the foundation for retry strategies, preventing thundering herd problems while giving temporary issues time to resolve. Payment-specific backoff must account for customer experience - users will abandon checkout flows that take too long.

Operation Type	Initial Delay	Max Delay	Max Attempts	Jitter Strategy
Payment intent creation	100ms	5s	3 attempts	±25% random jitter
Charge processing	200ms	10s	4 attempts	±50% random jitter
3DS confirmation	500ms	30s	6 attempts	±25% random jitter (user waiting)
Webhook processing	1s	5 minutes	8 attempts	Exponential jitter (background task)
Refund processing	1s	1 minute	5 attempts	±25% random jitter
Reconciliation queries	5s	10 minutes	12 attempts	Full jitter (low priority)

The retry schedule must consider cumulative delay - if initial payment processing allows 30 seconds total, the retry schedule cannot exceed this budget even for multiple failed attempts.

Jitter calculation prevents synchronized retries:

- **Random jitter:** delay \pm (percentage * delay)
- **Exponential jitter:** random value between 0 and calculated exponential delay
- **Full jitter:** completely random delay between minimum and maximum bounds

Circuit Breaker Pattern

Circuit breakers protect payment systems from cascading failures by temporarily blocking requests to failing downstream services. Payment circuit breakers must be carefully tuned to avoid disrupting legitimate transactions while protecting system stability.

Circuit Breaker Configuration	Failure Threshold	Recovery Time	Half-Open Test Requests
Payment provider API calls	10 failures in 1 minute	30 seconds	3 requests
Webhook signature verification	20 failures in 5 minutes	60 seconds	5 requests
Database connections	5 failures in 30 seconds	10 seconds	2 requests
3DS authentication service	15 failures in 2 minutes	45 seconds	2 requests
Reconciliation API calls	25 failures in 10 minutes	5 minutes	10 requests

Circuit breaker states require careful monitoring and alerting:

- Closed State:** Normal operation, requests flow through normally, failure counting active
- Open State:** All requests immediately fail without calling downstream service, preventing cascade failures
- Half-Open State:** Limited test requests allowed to check if downstream service has recovered

Design Principle: Circuit breakers should fail fast and fail visibly - when payments cannot be processed due to circuit breaker activation, users should receive clear messaging about temporary unavailability rather than generic errors.

Retry Decision Matrix

Different error types require different retry strategies based on their likelihood of resolution and potential for success. The retry decision matrix codifies these strategies for consistent application across the payment system.

Error Category	Retry Immediately	Retry with Backoff	No Retry	Special Handling
Network timeout	✓ (1 immediate)	✓ (exponential)	After max attempts	Query provider status
HTTP 5xx errors	✓ (1 immediate)	✓ (exponential)	After max attempts	Circuit breaker activation
Rate limiting (HTTP 429)	X	✓ (honor retry-after)	If rate limit exceeded	Token bucket implementation
Authentication errors	X	X	✓ (permanent failure)	Alert operations team
Invalid request data	X	X	✓ (client error)	Return validation errors
Insufficient funds	X	X	✓ (business rule)	Update payment method
Provider maintenance	X	✓ (longer delays)	After extended period	Consider backup provider
Database deadlock	✓ (1 immediate)	✓ (short backoff)	After 3 attempts	Different retry pattern

The retry decision must be made quickly to maintain acceptable response times. Pre-calculated retry matrices enable immediate classification without complex decision logic in the hot path.

Background Retry Processing

Some payment operations benefit from background retry processing, particularly webhook handling and reconciliation tasks that don't require immediate user feedback. Background retries can use more aggressive strategies and longer timeframes.

Background retry processing follows a job queue pattern with the following characteristics:

1. **Immediate Queue:** Failed operations retried within seconds for transient issues
2. **Delayed Queue:** Operations retried after minutes or hours for provider outages
3. **Manual Queue:** Operations requiring human intervention or investigation

Queue Type	Retry Schedule	Max Age	Success Criteria
Immediate webhook retry	1s, 2s, 4s, 8s, 16s	5 minutes	HTTP 200 + valid response
Delayed webhook retry	1min, 5min, 15min, 1hr	24 hours	Successful processing
Reconciliation retry	5min, 15min, 1hr, 4hr	7 days	Status matches provider
Refund retry	30s, 2min, 10min, 1hr	6 hours	Refund confirmed
Manual investigation	No automatic retry	30 days	Human resolution

Background retry queues require dead letter handling for operations that exhaust retry attempts. Dead letter queues should trigger alerting and provide audit trails for manual investigation.

Data Consistency Guarantees

Payment systems require stronger consistency guarantees than typical web applications because financial data must be accurate and auditable. The challenge lies in maintaining consistency across multiple external systems while providing acceptable performance for real-time payment processing.

Think of payment data consistency like maintaining accurate bank ledgers before computerization - every transaction must be recorded completely and correctly, with clear audit trails and reconciliation processes to catch and correct any discrepancies. Modern payment systems face the additional challenge of coordinating these "ledger entries" across multiple distributed systems that may fail independently.

ACID Compliance for Payment Operations

Payment operations must maintain ACID properties within database boundaries while implementing compensation patterns for operations that span multiple systems. The key insight is identifying which operations can be truly atomic versus those requiring eventual consistency with reconciliation.

Atomic Operations (Single Database Transaction):

Operation	ACID Guarantees	Transaction Scope
Payment intent creation with idempotency	Full ACID within single DB transaction	Intent record + idempotency key record
Charge status updates from webhooks	Full ACID for status transitions	Charge record + webhook event record
Refund record creation	Full ACID for refund tracking	Refund record + charge balance updates
Payment method tokenization	Full ACID for token storage	Token record + customer association

Eventually Consistent Operations (Cross-System Coordination):

Operation	Consistency Strategy	Reconciliation Method
Charge creation at payment provider	Compensating transaction on failure	Status polling + webhook confirmation
3DS authentication completion	Status reconciliation after timeout	Provider query + local state update
Webhook event processing	Idempotent processing with deduplication	Replay missed events from provider
Multi-provider payment routing	Distributed saga pattern	Cancel backup providers on primary success

Decision: Database Transaction Boundaries

- **Context:** Payment operations often require coordination between local database updates and external provider API calls
- **Options Considered:**
 1. Large transactions encompassing external API calls
 2. Local database transactions only with external compensation
 3. Two-phase commit with payment providers
- **Decision:** Local database transactions with external compensation and reconciliation
- **Rationale:** Payment providers don't support 2PC, and large transactions risk timeouts and deadlocks during network calls
- **Consequences:** Requires robust reconciliation but provides better availability and clearer failure modes

Idempotency as Consistency Foundation

Idempotency serves as the foundational consistency mechanism for payment systems, ensuring that duplicate operations produce identical results rather than corrupting financial data. Every operation that moves money or changes payment state must be idempotent.

The idempotency implementation extends beyond simple duplicate prevention to encompass complete operation replay capability. When network failures create uncertainty about operation completion, idempotency allows safe replay of the entire operation sequence.

Operation Type	Idempotency Key Source	Duplicate Behavior
Payment intent creation	Client-provided idempotency key	Return existing intent if parameters match
Charge processing	Payment intent ID + payment method hash	Return existing charge if successful
Refund creation	Client-provided idempotency key + charge ID	Return existing refund for same key
Webhook processing	Provider event ID + event type	Skip processing if already handled
3DS confirmation	Provider payment intent ID	Return current authentication status
Reconciliation run	Date range + run type hash	Return previous run results

Idempotency key validation must prevent parameter tampering - using the same idempotency key with different payment parameters should result in an error, not returning the previous result with different parameters.

Idempotency Key Lifecycle Management:

1. **Key Generation:** Client generates UUID v4 or similar high-entropy identifier
2. **Parameter Hashing:** System creates hash of essential operation parameters
3. **Conflict Detection:** Same key with different parameters triggers validation error
4. **Response Caching:** Successful operation response cached for key lifetime
5. **Expiration:** Keys expire after 24 hours to prevent indefinite storage growth

Reconciliation as Consistency Safety Net

Reconciliation provides the ultimate consistency guarantee by periodically comparing local payment state against authoritative provider records. This catches issues that escape other consistency mechanisms and provides audit trails for financial reporting.

The reconciliation strategy operates on multiple timescales to balance consistency with system load:

Real-Time Reconciliation (Immediate):

- Triggered by webhook processing failures
- Validates critical state transitions (payment success/failure)
- Limited to individual payment intent scope
- Must complete within seconds to support user experience

Scheduled Reconciliation (Periodic):

Reconciliation Type	Frequency	Scope	Discrepancy Handling
Recent payments	Every 15 minutes	Last 4 hours of payments	Immediate correction + alerting
Daily settlement	Every 24 hours	Previous day's completed payments	Batch correction + audit log
Weekly audit	Every 7 days	All payments from previous week	Full investigation required
Monthly compliance	Every 30 days	Complete payment history	Regulatory reporting + correction

Reconciliation algorithms must handle the complexity of timing differences between local and provider systems. Providers may process refunds or chargebacks that haven't been communicated via webhooks yet, requiring sophisticated

discrepancy analysis.

Discrepancy Resolution Workflow:

1. **Detection:** Automated comparison identifies state differences
2. **Classification:** Discrepancy categorized by type and severity
3. **Investigation:** Automated analysis determines probable cause
4. **Resolution:** Appropriate correction applied based on discrepancy type
5. **Verification:** Correction verified through subsequent reconciliation
6. **Audit:** All discrepancies and resolutions logged for compliance

Discrepancy Type	Auto-Resolution	Manual Review Required
Missing webhook events	✓	Log for pattern analysis
Amount mismatches	✗	Always require human review
Status transitions	✓	Only for expected transitions
Refund timing differences	✓	If within expected settlement window
Unknown charges at provider	✗	Potential fraud investigation
Extra local charges	✗	Data corruption investigation

Critical Insight: Reconciliation is not just error correction - it's also fraud detection. Unexpected discrepancies between local and provider state can indicate security breaches or system compromise.

Compensation Transaction Patterns

When operations span multiple systems and atomic transactions are impossible, compensation transactions provide consistency by implementing logical rollback through inverse operations. Payment systems require sophisticated compensation because financial operations often cannot be simply undone.

Compensation Strategies by Operation Type:

Failed Operation	Compensation Action	Timing Constraints
Provider charge succeeded, local DB update failed	Update local charge status to match provider	Must complete before user notification
Provider charge failed, local charge marked succeeded	Mark local charge as failed, update intent status	Immediate - prevent double charging
Refund created locally but provider call failed	Cancel local refund, restore charge balance	Before user notification of refund
3DS authentication completed but confirmation failed	Query provider status, update local accordingly	Within authentication window (10-15 minutes)
Webhook processing partially completed	Replay webhook with idempotency protection	Background processing acceptable

Compensation transactions must be idempotent themselves, as they may be replayed during recovery scenarios. The compensation logic should also be reversible in case the compensation itself needs to be undone based on new information.

Compensation Transaction Ordering:

1. **Immediate Compensation:** Fix issues that could cause immediate user impact or double charging
2. **Background Compensation:** Handle provider synchronization and reconciliation
3. **Audit Compensation:** Log all compensation actions for regulatory compliance and debugging

The compensation system requires comprehensive logging to track the sequence of operations and compensations, enabling debugging of complex failure scenarios and providing audit trails for financial compliance.

Implementation Guidance

The error handling implementation transforms the theoretical failure analysis into concrete Python code that provides robust payment processing even under adverse conditions. This implementation provides complete error handling infrastructure that spans all payment system components.

Technology Recommendations

Component	Simple Option	Advanced Option
Retry Library	<code>tenacity</code> with exponential backoff	Custom retry framework with circuit breakers
Circuit Breaker	<code>pybreaker</code> library	<code>circuitbreaker</code> with custom metrics
Background Jobs	<code>celery</code> with Redis broker	<code>RQ</code> with Redis or database broker
Monitoring	<code>structlog + prometheus_client</code>	<code>OpenTelemetry</code> with distributed tracing
Error Tracking	<code>sentry-sdk</code> for error aggregation	Custom error dashboard with <code>Grafana</code>
Database Retries	<code>SQLAlchemy</code> with connection pooling	<code>asyncpg</code> with custom retry logic

Project Structure Integration

```
payment-gateway/
  src/
    payment_gateway/
      error_handling/
        __init__.py           ← Error handling public interface
        retry_strategies.py   ← Exponential backoff and retry logic
        circuit_breakers.py   ← Circuit breaker implementations
        consistency.py         ← Data consistency helpers
        reconciliation.py     ← State reconciliation logic
        compensation.py       ← Compensation transaction handlers
        exceptions.py         ← Custom exception hierarchy
      core/
        payment_intents.py    ← Uses retry strategies for provider calls
        charges.py             ← Uses circuit breakers for external APIs
        webhooks.py            ← Uses reconciliation for state sync
      infrastructure/
        providers.py          ← Provider API clients with error handling
        database.py            ← Database connection with retry logic
        monitoring.py          ← Error metrics and alerting
  tests/
    error_handling/
      test_retry_strategies.py  ← Retry behavior validation
      test_circuit_breakers.py  ← Circuit breaker state transitions
      test_reconciliation.py    ← Reconciliation accuracy tests
```

Infrastructure Code: Retry Strategy Framework

```
"""
Complete retry strategy implementation with exponential backoff and jitter.

This provides the foundation for all payment operation retries.

"""

import asyncio
import logging
import random
import time
from dataclasses import dataclass
from enum import Enum
from typing import Any, Callable, Optional, Type, Union
from functools import wraps

logger = logging.getLogger(__name__)

class RetryStrategy(Enum):
    EXPONENTIAL = "exponential"
    FIXED = "fixed"
    LINEAR = "linear"

    @dataclass
    class RetryConfig:
        max_attempts: int
        initial_delay: float
        max_delay: float
        strategy: RetryStrategy
        jitter: bool = True
        exponential_base: float = 2.0
        retryable_exceptions: tuple = (Exception,)
```

PYTHON

```
class RetryExhaustedException(Exception):

    """Raised when all retry attempts are exhausted."""

    def __init__(self, attempts: int, last_exception: Exception):
        self.attempts = attempts
        self.last_exception = last_exception
        super().__init__(f"Retry exhausted after {attempts} attempts: {last_exception}")

def calculate_delay(attempt: int, config: RetryConfig) -> float:

    """Calculate delay for given attempt number using configured strategy."""

    if config.strategy == RetryStrategy.EXponential:
        delay = config.initial_delay * (config.exponential_base ** (attempt - 1))
    elif config.strategy == RetryStrategy.LINEAR:
        delay = config.initial_delay * attempt
    else: # FIXED
        delay = config.initial_delay

    delay = min(delay, config.max_delay)

    if config.jitter:
        # Add ±25% jitter to prevent thundering herd
        jitter_amount = delay * 0.25
        delay += random.uniform(-jitter_amount, jitter_amount)

    return max(0, delay)

def retry_with_config(config: RetryConfig):

    """Decorator factory for adding retry logic to functions."""

    def decorator(func: Callable):
        @wraps(func)
        def wrapper(*args, **kwargs):
            last_exception = None
```

```
for attempt in range(1, config.max_attempts + 1):

    try:

        return func(*args, **kwargs)

    except config.retryable_exceptions as e:

        last_exception = e

        if attempt == config.max_attempts:

            logger.error(f"Function {func.__name__} failed after {attempt} attempts",
                         exc_info=True)

            raise RetryExhaustedException(attempt, e)

        delay = calculate_delay(attempt, config)

        logger.warning(f"Function {func.__name__} attempt {attempt} failed, "
                      f"retrying in {delay:.2f}s: {e}")

        time.sleep(delay)

    except Exception as e:

        # Non-retryable exception, fail immediately

        logger.error(f"Function {func.__name__} failed with non-retryable error",
                     exc_info=True)

        raise

# This should never be reached, but just in case

raise RetryExhaustedException(config.max_attempts, last_exception)

return wrapper

return decorator

# Async version for async functions

def async_retry_with_config(config: RetryConfig):
```

```
"""Async decorator factory for adding retry logic to async functions."""

def decorator(func: Callable):

    @wraps(func)

    async def wrapper(*args, **kwargs):

        last_exception = None

        for attempt in range(1, config.max_attempts + 1):

            try:

                return await func(*args, **kwargs)

            except config.retryable_exceptions as e:

                last_exception = e

                if attempt == config.max_attempts:

                    logger.error(f"Async function {func.__name__} failed after {attempt} attempts",
                                exc_info=True)

                    raise RetryExhaustedException(attempt, e)

                delay = calculate_delay(attempt, config)

                logger.warning(f"Async function {func.__name__} attempt {attempt} failed, "
                               f"retrying in {delay:.2f}s: {e}")

                await asyncio.sleep(delay)

            except Exception as e:

                # Non-retryable exception, fail immediately

                logger.error(f"Async function {func.__name__} failed with non-retryable error",
                                exc_info=True)

                raise

        raise RetryExhaustedException(config.max_attempts, last_exception)
```

```
    return wrapper

    return decorator

# Pre-configured retry strategies for common payment operations

PAYMENT_INTENT_RETRY = RetryConfig(
    max_attempts=3,
    initial_delay=0.1,
    max_delay=5.0,
    strategy=RetryStrategy.EXponential,
    jitter=True
)

CHARGE_PROCESSING_RETRY = RetryConfig(
    max_attempts=4,
    initial_delay=0.2,
    max_delay=10.0,
    strategy=RetryStrategy.EXponential,
    jitter=True
)

WEBHOOK_PROCESSING_RETRY = RetryConfig(
    max_attempts=8,
    initial_delay=1.0,
    max_delay=300.0, # 5 minutes
    strategy=RetryStrategy.Exponential,
    jitter=True
)
```

Infrastructure Code: Circuit Breaker Implementation

```
"""
Circuit breaker implementation for protecting payment systems from cascading failures.

"""

import time
import threading

from dataclasses import dataclass

from enum import Enum

from typing import Any, Callable, Optional

from functools import wraps

class CircuitState(Enum):
    CLOSED = "closed"
    OPEN = "open"
    HALF_OPEN = "half_open"

    @dataclass
    class CircuitBreakerConfig:
        failure_threshold: int
        recovery_timeout: float
        expected_exception: tuple = (Exception, )
        half_open_max_calls: int = 3

    class CircuitBreakerOpenException(Exception):
        """Raised when circuit breaker is open and calls are blocked."""

        pass

    class CircuitBreaker:
        """Thread-safe circuit breaker implementation."""

        def __init__(self, config: CircuitBreakerConfig):
```

PYTHON

```
    self.config = config

    self._state = CircuitState.CLOSED

    self._failure_count = 0

    self._last_failure_time = None

    self._half_open_calls = 0

    self._lock = threading.RLock()

    @property

    def state(self) -> CircuitState:

        with self._lock:

            return self._state

    def _should_attempt_reset(self) -> bool:

        """Check if enough time has passed to attempt reset from open to half-open."""

        return (self._last_failure_time and

                time.time() - self._last_failure_time >= self.config.recovery_timeout)

    def _reset_to_closed(self):

        """Reset circuit breaker to closed state."""

        self._state = CircuitState.CLOSED

        self._failure_count = 0

        self._half_open_calls = 0

        self._last_failure_time = None

    def _record_success(self):

        """Record successful call."""

        with self._lock:

            if self._state == CircuitState.HALF_OPEN:

                self._reset_to_closed()
```

```

def _record_failure(self):

    """Record failed call and update circuit state."""

    with self._lock:

        self._failure_count += 1

        self._last_failure_time = time.time()

    if self._state == CircuitState.CLOSED:

        if self._failure_count >= self.config.failure_threshold:

            self._state = CircuitState.OPEN

    elif self._state == CircuitState.HALF_OPEN:

        self._state = CircuitState.OPEN


def call(self, func: Callable, *args, **kwargs) -> Any:

    """Execute function call through circuit breaker."""

    with self._lock:

        if self._state == CircuitState.OPEN:

            if self._should_attempt_reset():

                self._state = CircuitState.HALF_OPEN

                self._half_open_calls = 0

            else:

                raise CircuitBreakerOpenException(
                    f"Circuit breaker is open. Retry after {self.config.recovery_timeout}"
                )

        if self._state == CircuitState.HALF_OPEN:

            if self._half_open_calls >= self.config.half_open_max_calls:

                raise CircuitBreakerOpenException(
                    "Circuit breaker half-open call limit reached"
                )

            self._half_open_calls += 1

```

```
try:

    result = func(*args, **kwargs)

    self._record_success()

    return result

except self.config.expected_exception as e:

    self._record_failure()

    raise

except Exception as e:

    # Unexpected exception doesn't affect circuit breaker

    raise


def circuit_breaker(config: CircuitBreakerConfig):

    """Decorator factory for adding circuit breaker to functions."""

    breaker = CircuitBreaker(config)


    def decorator(func: Callable):

        @wraps(func)

        def wrapper(*args, **kwargs):

            return breaker.call(func, *args, **kwargs)

        return wrapper

    return decorator


# Pre-configured circuit breakers for payment operations

PROVIDER_API_BREAKER = CircuitBreakerConfig(

    failure_threshold=10,

    recovery_timeout=30.0,

    half_open_max_calls=3

)

DATABASE_BREAKER = CircuitBreakerConfig(
```

```
failure_threshold=5,  
recovery_timeout=10.0,  
half_open_max_calls=2  
)
```

Core Logic Skeleton: Error Handling Integration

```
"""
Core error handling integration with payment processing operations.

Skeleton code showing how to integrate retry strategies and circuit breakers.

"""

from typing import Dict, Any, Optional

from payment_gateway.error_handling.retry_strategies import (
    retry_with_config, CHARGE_PROCESSING_RETRY, WEBHOOK_PROCESSING_RETRY
)

from payment_gateway.error_handling.circuit_breakers import (
    circuit_breaker, PROVIDER_API_BREAKER, CircuitBreakerOpenException
)

from payment_gateway.error_handling.exceptions import (
    PaymentProcessingError, ProviderError, ReconciliationError
)

class PaymentProcessor:

    """Core payment processing with comprehensive error handling."""

    @retry_with_config(CHARGE_PROCESSING_RETRY)
    @circuit_breaker(PROVIDER_API_BREAKER)

    def create_charge(self, payment_method_token: str, amount: int,
                      currency: str, return_url: str, metadata: Dict[str, Any]) -> Dict[str, Any]:
        """
        Create charge with provider using retry and circuit breaker protection.

        TODO 1: Validate input parameters (amount > 0, valid currency, etc.)
        TODO 2: Prepare charge request payload with idempotency key
        TODO 3: Make HTTP request to provider API with timeout
        TODO 4: Handle provider-specific error codes and map to internal taxonomy
        """

    
```

```
    TODO 5: Parse successful response and extract charge details
```

```
    TODO 6: Return normalized charge response
```

```
Raises:
```

```
    PaymentProcessingError: For business logic violations
```

```
    ProviderError: For provider API failures (retryable)
```

```
    CircuitBreakerOpenException: When provider is unavailable
```

```
    """
```

```
pass
```

```
@retry_with_config(WEBHOOK_PROCESSING_RETRY)
```

```
def process_webhook_event(self, event_type: str, payload: Dict[str, Any],  
                           signature: str, source_ip: str) -> None:
```

```
    """
```

```
Process webhook event with retry protection for transient failures.
```

```
    TODO 1: Verify webhook signature using HMAC validation
```

```
    TODO 2: Check for duplicate event processing using deduplication key
```

```
    TODO 3: Route event to type-specific handler based on event_type
```

```
    TODO 4: Update local payment state within database transaction
```

```
    TODO 5: Handle state transition validation and conflicts
```

```
    TODO 6: Log successful processing for audit trail
```

```
Raises:
```

```
    WebhookProcessingError: For invalid webhooks or processing failures
```

```
    ReconciliationError: For state consistency violations
```

```
    """
```

```
pass
```

```
def handle_provider_error(self, error_code: str, error_message: str,
```

```
        operation: str) -> None:

"""

Handle provider-specific errors with appropriate retry classification.

TODO 1: Map provider error code to internal error taxonomy

TODO 2: Determine if error is retryable or terminal

TODO 3: Log error details with appropriate severity level

TODO 4: Update operation status based on error type

TODO 5: Trigger alerting for non-transient errors

TODO 6: Return appropriate error response to client

"""

pass


def reconcile_payment_state(self, payment_intent_id: str) -> None:

"""

Reconcile local payment state with provider state.

TODO 1: Query provider API for current payment status

TODO 2: Compare provider state with local database records

TODO 3: Identify discrepancies in status, amounts, or timestamps

TODO 4: Classify discrepancies by type and severity

TODO 5: Apply automatic corrections for expected discrepancy types

TODO 6: Flag manual review required for unexpected discrepancies

TODO 7: Log all reconciliation actions for audit trail

"""

pass


def execute_compensation_transaction(self, failed_operation: str,
                                      context: Dict[str, Any]) -> None:

"""
```

```
Execute compensation transaction to maintain data consistency.
```

```
TODO 1: Identify compensation strategy based on failed operation type  
TODO 2: Check current system state to ensure compensation is still needed  
TODO 3: Execute compensation within database transaction  
TODO 4: Verify compensation completed successfully  
TODO 5: Log compensation action with full context  
TODO 6: Schedule follow-up reconciliation to verify consistency
```

```
Note: This function must be idempotent since compensations may be retried
```

```
"""
```

```
pass
```

Milestone Checkpoint: Error Handling Validation

After implementing the error handling infrastructure, validate functionality with these checkpoints:

Retry Strategy Validation:

```
# Test exponential backoff behavior  
  
python -m pytest tests/error_handling/test_retry_strategies.py -v  
  
# Expected output should show:  
  
# - Retry attempts with increasing delays  
# - Jitter preventing synchronized retries  
# - Proper exception handling for terminal vs retryable errors  
# - Configuration validation for different retry strategies
```

BASH

Circuit Breaker Validation:

```
# Test circuit breaker state transitions

python -m pytest tests/error_handling/test_circuit_breakers.py -v

# Expected output should show:

# - Closed -> Open transition after failure threshold

# - Half-open state allowing limited test calls

# - Open -> Half-open transition after recovery timeout

# - Proper exception handling when circuit is open
```

BASH

Integration Test:

```
# Test full payment flow with error injection

python scripts/test_error_scenarios.py --scenario network_timeout

# Should demonstrate:

# - Payment processing continues despite network issues

# - Retry strategies prevent permanent failures for transient issues

# - Circuit breakers protect system during provider outages

# - Reconciliation catches and corrects state inconsistencies
```

BASH

Manual Validation Steps:

1. **Simulate provider timeout:** Configure test environment with delayed responses, verify retry behavior
2. **Test circuit breaker activation:** Send requests that consistently fail, verify circuit opens and prevents cascading failures
3. **Validate reconciliation:** Create state discrepancies in test database, run reconciliation, verify corrections
4. **Check compensation logic:** Simulate partial failures, verify compensation transactions restore consistency

Debugging Tips for Error Handling

Symptom	Likely Cause	Diagnosis Steps	Fix
Payments timing out frequently	Retry configuration too aggressive	Check retry delay calculations, monitor total processing time	Reduce max attempts or increase delay bounds
Circuit breaker opens unnecessarily	Failure threshold too low	Review error logs for actual vs expected failure rate	Increase failure threshold or improve error classification
Duplicate payment processing	Idempotency key conflicts	Check idempotency key generation and storage	Fix key generation algorithm, add parameter validation
Reconciliation shows persistent discrepancies	Webhook processing failures	Review webhook logs for signature validation and processing errors	Fix webhook signature validation, replay missed events
Memory usage growing over time	Retry state not being cleaned up	Monitor retry attempt tracking and circuit breaker state	Add cleanup for expired retry attempts and old circuit breaker state
Database deadlocks during error recovery	Compensation transactions conflicting	Check transaction isolation levels and lock ordering	Implement consistent lock ordering, use shorter transactions

Testing Strategy

Milestone(s): All milestones - comprehensive testing strategy covers payment intent & idempotency (Milestone 1), payment processing & 3DS (Milestone 2), refunds & disputes (Milestone 3), and webhook reconciliation (Milestone 4)

Testing a payment processing system presents unique challenges that go far beyond typical web application testing. Think of testing a payment gateway like testing a bank's ATM network - you need to verify not just that money moves correctly under normal conditions, but that the system maintains integrity during network partitions, handles fraud attempts gracefully, and never loses track of a single cent even when external systems fail unexpectedly. The stakes are high: a bug that allows duplicate charges could cost thousands in chargebacks, while a vulnerability that exposes card data could trigger PCI compliance violations and regulatory penalties.

The mental model for payment system testing combines three distinct testing philosophies. First, **state machine verification** ensures payment intents transition through their lifecycle correctly and can never reach invalid states. Second, **external dependency simulation** validates behavior when payment providers are slow, unavailable, or return unexpected responses. Third, **consistency verification** confirms that money tracking remains accurate across all possible failure scenarios, including partial network failures and race conditions.

Payment systems operate in an eventually consistent distributed environment where webhooks can arrive out of order, network requests can timeout after succeeding on the provider side, and external systems can enter maintenance mode during critical payment flows. Your testing strategy must account for these realities while providing confidence that the core business logic - moving money safely and accurately - remains bulletproof under all conditions.

Core Test Scenarios

The foundation of payment system testing rests on comprehensive scenario coverage that spans happy path flows, systematic error injection, and edge case validation. Each test scenario maps to specific business risks and technical failure modes that could compromise payment integrity or user experience.

Happy Path Test Categories

Payment Intent Creation and Management testing validates the core payment intent lifecycle from creation through completion. These tests verify that payment intents progress through their state machine correctly, that idempotency keys prevent duplicate processing, and that intent expiration cleans up stale records appropriately.

Test Scenario	Validation Points	Expected Behavior	Risk Mitigated
Create payment intent with valid parameters	Intent created with correct status, metadata stored, client secret generated	<code>PaymentIntent</code> record with status <code>created</code> , unique <code>client_secret</code> , stored <code>metadata</code>	Invalid payment intent creation
Create intent with existing idempotency key	Same response returned, no duplicate intent created	Existing <code>PaymentIntent</code> returned, same <code>client_secret</code>	Duplicate charge processing
Create intent with reused key but different params	Error response indicating parameter mismatch	HTTP 400 with specific error about idempotency key conflict	Idempotency key abuse
Process payment intent expiration	Expired intents marked as canceled, cleanup completed	Status transitions to <code>canceled</code> , database cleanup executed	Resource leaks from stale intents

Payment Processing Flow testing ensures that charge creation, 3D Secure authentication, and payment confirmation work correctly across different payment methods and issuer requirements. These scenarios validate the complete payment journey from payment method tokenization through final settlement.

Test Scenario	Validation Points	Expected Behavior	Risk Mitigated
Process card payment without 3DS	Charge created and immediately succeeds	<code>Charge</code> with status <code>succeeded</code> , settlement details recorded	Basic payment processing failure
Process card payment requiring 3DS	Authentication challenge initiated correctly	<code>Charge</code> status <code>pending</code> , 3DS redirect URL provided	3DS flow misconfiguration
Complete 3DS authentication successfully	Payment confirmed after authentication	<code>Charge</code> status <code>succeeded</code> , authentication details stored	3DS completion failure
Handle 3DS authentication timeout	Payment fails gracefully with timeout error	<code>Charge</code> status <code>failed</code> , appropriate error message	3DS timeout handling
Process payment with invalid card	Validation errors returned appropriately	<code>Charge</code> status <code>failed</code> , specific decline reason provided	Invalid payment method handling

Error Condition Test Categories

Network and Provider Failure testing simulates various failure modes that occur when communicating with external payment providers. These tests ensure the system handles timeouts, service unavailability, and malformed responses without losing payment state or creating inconsistencies.

Failure Scenario	Injection Method	Expected System Behavior	Recovery Validation
Provider API timeout during charge creation	Mock provider returns timeout after delay	Charge remains in pending status, retry scheduled	Retry logic executes with exponential backoff
Provider returns 5xx error	Mock returns HTTP 500 for charge requests	Charge creation fails with retriable error	Circuit breaker triggers after threshold
Network partition during 3DS confirmation	Connection drop simulation during confirm call	Payment state preserved, retry attempted	Reconciliation detects and resolves state mismatch
Malformed webhook payload received	Send invalid JSON to webhook endpoint	Webhook rejected, no state changes made	Signature verification prevents processing
Provider webhook signature mismatch	Send webhook with incorrect HMAC signature	Webhook rejected, security event logged	Prevents webhook spoofing attacks

Database and Persistence Failure testing validates behavior when the underlying data store experiences issues. These scenarios ensure that payment operations maintain ACID properties and that the system can recover gracefully from storage failures.

Database Failure Mode	Test Setup	Expected Behavior	Consistency Check
Transaction rollback during payment creation	Force database constraint violation	Payment intent creation fails cleanly	No partial records created
Connection loss during charge processing	Disconnect database mid-transaction	Operation fails, no state corruption	Idempotency allows safe retry
Deadlock during concurrent payment processing	Submit simultaneous payments for same intent	One succeeds, others receive conflict error	Payment intent processed exactly once
Storage full during webhook processing	Fill disk space during event storage	Webhook processing deferred, not lost	Events reprocessed after space recovery

Edge Case Test Categories

Race Condition and Concurrency testing exposes timing-dependent bugs that occur when multiple operations interact with the same payment resources simultaneously. These scenarios are critical for ensuring payment integrity under load.

Race Condition Scenario	Test Implementation	Expected Resolution	Data Integrity Check
Concurrent payment processing attempts	Submit multiple process requests simultaneously	First request succeeds, others return conflict	Payment intent processed exactly once
Webhook arrival before payment completion	Send success webhook before local charge completes	Webhook processed, local state synchronized	Final state consistent across systems
Simultaneous refund requests	Submit multiple refund requests for same charge	First refund succeeds, others return insufficient funds	Refund amount never exceeds original charge
Payment cancellation during processing	Cancel payment intent while charge is processing	Cancellation queued, processing completes first	Terminal state determined correctly

Data Validation and Security testing ensures that the system properly validates all inputs and maintains security boundaries throughout the payment flow. These tests prevent injection attacks, data corruption, and unauthorized access to sensitive payment information.

Security Test Category	Attack Vector	Validation Method	Protection Verified
SQL injection in payment parameters	Malicious SQL in payment metadata	Parameters properly escaped/parameterized	Database integrity maintained
XSS in webhook payload processing	JavaScript injection in webhook events	Content properly sanitized	No script execution
Payment amount manipulation	Negative amounts, overflow values	Amount validation prevents processing	Invalid amounts rejected
Idempotency key exhaustion	Generate excessive unique keys	Rate limiting and key expiration	Resource exhaustion prevented
Webhook signature bypass	Various signature manipulation attempts	All invalid signatures rejected	Webhook authenticity guaranteed

Testing Philosophy: Fail-Safe Defaults

Every test scenario should validate that failures result in safe default behavior. When in doubt, the system should reject transactions rather than process them incorrectly. A false negative (declining a valid payment) is always preferable to a false positive (accepting an invalid payment) in financial systems.

Milestone Validation Checkpoints

Each milestone introduces specific capabilities that require targeted validation to ensure proper implementation before proceeding to dependent functionality. These checkpoints provide concrete verification steps that confirm not just functional correctness, but also non-functional requirements like performance, security, and reliability.

Milestone 1: Payment Intent & Idempotency Validation

The first milestone establishes the foundation for all payment processing by implementing payment intent creation with robust idempotency controls. Validation focuses on state management correctness and duplicate prevention mechanisms.

Functional Validation Checkpoints:

Checkpoint	Test Command	Expected Output	Failure Investigation
Payment intent creation	<code>curl -X POST /payment-intents -d '{"amount": 1000, "currency": "usd"}' -H "Content-Type: application/json"</code>	HTTP 200, JSON response with <code>id</code> , <code>client_secret</code> , status <code>created</code>	Check database constraints, validate amount conversion
Idempotency key handling	Same request with <code>Idempotency-Key: test-key-123</code> header twice	First returns 200 with new intent, second returns same response	Verify idempotency key storage, check for duplicate records
Parameter validation	<code>curl</code> with invalid currency code "INVALID"	HTTP 400 with specific validation error message	Check validation logic, error message formatting
Intent expiration	Create intent, wait past expiration time, run cleanup job	Intent status changes to <code>canceled</code> , expired intents removed	Verify cleanup job scheduling, status transition logic

State Machine Validation:

The payment intent state machine must enforce valid transitions and prevent invalid state changes. This validation ensures that intents cannot reach corrupted states that would break downstream processing.

Current State	Valid Transitions	Test Method	Expected Behavior
<code>created</code>	→ <code>processing</code> , <code>canceled</code> , <code>requires_action</code>	Direct database state updates	Transitions succeed, invalid transitions rejected
<code>processing</code>	→ <code>succeeded</code> , <code>failed</code> , <code>requires_action</code>	Mock payment processing outcomes	State updates correctly based on processing result
<code>requires_action</code>	→ <code>processing</code> , <code>canceled</code>	Simulate 3DS completion or timeout	Authentication flow updates state appropriately
<code>succeeded</code>	No transitions allowed (terminal)	Attempt to modify succeeded intent	All modification attempts rejected
<code>canceled</code>	No transitions allowed (terminal)	Attempt to process canceled intent	Processing requests return appropriate error

Data Consistency Validation:

Payment intent data must remain consistent across all system components, with proper referential integrity and audit trail maintenance.

```
# Database consistency check script

python scripts/validate_milestone1.py --check-consistency

# Expected output:

# ✓ All payment intents have valid status values

# ✓ All idempotency keys have corresponding intents

# ✓ No orphaned idempotency records found

# ✓ All client secrets are properly formatted UUIDs

# ✓ Intent metadata validates against schema
```

BASH

Milestone 2: Payment Processing & 3DS Validation

The second milestone adds payment execution capabilities with 3D Secure authentication support. Validation emphasizes provider integration correctness and authentication flow reliability.

Payment Processing Flow Validation:

Processing Stage	Validation Command	Success Criteria	Debugging Steps
Payment method tokenization	<code>POST /payment-intents/{id}/process</code> with card details	Returns payment method token, no raw card data stored	Check tokenization API calls, verify PCI compliance
Charge creation	Process payment with valid tokenized method	<code>Charge</code> record created with <code>pending</code> status	Review provider API logs, check request formatting
3DS authentication	Submit payment requiring Strong Customer Authentication	Returns authentication challenge URL	Validate 3DS detection logic, check issuer requirements
Authentication completion	Complete 3DS flow with valid authentication	<code>Charge</code> status updates to succeeded	Verify authentication validation, check provider callbacks

3D Secure Flow Integration:

The 3D Secure authentication flow involves multiple redirects and callbacks that must be validated end-to-end to ensure users can complete authentication successfully.

3DS Flow Stage	Test Scenario	Expected Behavior	Error Handling
Challenge detection	Process payment requiring 3DS	Returns <code>requires_action</code> with challenge parameters	Non-3DS payments proceed directly
Authentication redirect	User redirects to issuer authentication page	Valid authentication URL with proper parameters	Invalid URLs cause graceful failure
Return URL handling	User completes authentication and returns	Payment processing resumes with authentication token	Malformed returns rejected safely
Authentication timeout	User abandons authentication flow	Payment fails with timeout after configured period	Timeout handling prevents resource leaks

Provider Integration Validation:

External payment provider integration must handle various response scenarios correctly while maintaining local state consistency.

```
# Provider integration test runner                                     PYTHON

python -m pytest tests/integration/payment_processing/ -v

# Expected test results:

# test_successful_payment_processing PASSED

# test_payment_requires_3ds_authentication PASSED

# test_payment_declined_by_issuer PASSED

# test_provider_timeout_handling PASSED

# test_network_error_retry_logic PASSED

# test_malformed_provider_response PASSED
```

Milestone 3: Refunds & Disputes Validation

The third milestone implements payment reversals through refunds and dispute handling mechanisms. Validation focuses on money movement accuracy and chargeback processing correctness.

Refund Processing Validation:

Refund Type	Test Scenario	Validation Points	Edge Case Handling
Full refund	Refund entire charge amount	Refund created with correct amount, charge marked as refunded	Cannot refund more than original amount
Partial refund	Refund portion of charge amount	Remaining balance tracked correctly, multiple partials allowed	Total refunds cannot exceed charge
Duplicate refund	Submit same refund request twice with idempotency	Second request returns existing refund, no double processing	Idempotency prevents over-refunding
Refund timing	Refund immediately vs. after settlement	Both scenarios processed correctly with appropriate provider calls	Settlement status affects refund type

Dispute and Chargeback Workflow:

Dispute handling requires integration with payment network notifications and evidence submission processes that must be validated against realistic chargeback scenarios.

Dispute Stage	Validation Method	Expected System Response	Manual Verification
Chargeback notification	Simulate webhook from payment provider	Dispute record created, status set to warning_needs_response	Check dispute reason code processing
Evidence submission	Submit dispute evidence through API	Evidence attached to dispute, status updated appropriately	Verify evidence formatting requirements
Dispute resolution	Process final dispute outcome webhook	Final status recorded, accounting entries balanced	Confirm win/loss handling accuracy

Money Movement Accuracy:

All refund and dispute operations must maintain accurate accounting with proper audit trails and balance reconciliation.

```
# Financial accuracy validation                                         BASH
python scripts/validate_milestone3.py --verify-accounting

# Expected accounting validation:

# ✓ All refunds sum to <= original charge amount

# ✓ Dispute fees properly recorded and tracked

# ✓ No orphaned refund records without parent charges

# ✓ All money movements have corresponding audit entries

# ✓ Balance calculations match provider records
```

Milestone 4: Webhook Reconciliation Validation

The final milestone implements reliable webhook processing with signature verification and state reconciliation. Validation emphasizes security, reliability, and eventual consistency guarantees.

Webhook Security Validation:

Security Aspect	Test Method	Pass Criteria	Security Implication
HMAC signature verification	Send webhooks with various signature values	Only valid signatures processed	Prevents webhook spoofing
Payload tampering detection	Modify webhook payload after signing	Modified payloads rejected	Ensures data integrity
Replay attack prevention	Resend old webhook with valid signature	Old webhooks rejected based on timestamp	Prevents replay attacks
Source IP validation	Send webhooks from unauthorized IP addresses	Unauthorized sources blocked	Additional security layer

Event Processing Reliability:

Webhook processing must handle various delivery scenarios including duplicates, out-of-order events, and processing failures while maintaining exactly-once processing semantics.

Processing Scenario	Test Implementation	Expected Behavior	Reliability Guarantee
Duplicate webhook delivery	Send identical webhook multiple times	First processes successfully, duplicates ignored	Exactly-once processing
Out-of-order event arrival	Send events with non-sequential timestamps	Events processed correctly regardless of arrival order	Temporal consistency maintained
Processing failure retry	Simulate processing failure during event handling	Failed events retry with exponential backoff	No events lost to transient failures
Webhook endpoint downtime	Disable webhook endpoint temporarily	Provider retries delivery, events processed after recovery	High availability maintained

State Reconciliation Accuracy:

Reconciliation processes must detect and resolve discrepancies between local payment state and external provider records while maintaining audit trails of all corrections.

```
# Reconciliation validation suite

python -m pytest tests/integration/reconciliation/ --verbose

# Key reconciliation test results:

# test_detect_local_vs_provider_discrepancies PASSED

# test_resolve_status_mismatches PASSED

# test_handle_missing_local_records PASSED

# test_process_provider_only_events PASSED

# test_reconciliation_audit_trail PASSED

# test_batch_reconciliation_performance PASSED
```

PYTHON

End-to-End Payment Journey Validation:

The complete payment flow from intent creation through webhook confirmation must work seamlessly across all implemented milestones.

E2E Test Scenario	Journey Steps	Success Metrics	Integration Points
Successful payment with 3DS	Create intent → Process payment → Complete 3DS → Receive webhook	Payment succeeds, all states consistent	All 4 milestones integrated
Failed payment with refund	Create intent → Process payment → Payment fails → Issue refund	Proper error handling, refund processed	Milestones 1, 2, 3 integrated
Disputed payment resolution	Successful payment → Chargeback webhook → Evidence submission → Resolution	Dispute handled correctly, accounting accurate	Milestones 1, 2, 3, 4 integrated

Validation Philosophy: Trust but Verify

While automated tests validate functional correctness, manual verification of critical flows ensures that the system behaves correctly from a user perspective. Each milestone should include manual testing scripts that simulate realistic user journeys and business scenarios.

Mock Payment Provider Setup

Testing payment systems requires sophisticated mock implementations that simulate external payment providers with realistic behaviors, error conditions, and timing characteristics. The mock provider must balance simplicity for development with fidelity to real provider behaviors for meaningful testing.

Mock Provider Architecture Decision

Decision: Configurable Mock Provider with Scenario Support

- Context:** Payment testing requires simulating various provider responses, error conditions, and timing behaviors without depending on external services or incurring transaction costs.
- Options Considered:**
 - Simple static responses for all requests
 - Configurable mock with scenario-based responses
 - Record-and-replay proxy to real provider
- Decision:** Configurable mock provider with scenario-based response control
- Rationale:** Provides deterministic testing while supporting complex scenarios like 3DS flows, timeouts, and error conditions. More reliable than proxy approach and more flexible than static responses.
- Consequences:** Requires maintenance to keep mock behavior aligned with real provider changes, but enables comprehensive testing without external dependencies.

The mock provider implementation follows a scenario-driven architecture where test cases configure expected behaviors before execution. This approach enables testing complex multi-step flows like 3D Secure authentication while maintaining predictable test outcomes.

Mock Provider Implementation Strategy

Core Mock Provider Components:

Component	Responsibility	Configuration Method	Behavioral Scope
Scenario Manager	Maintains test scenario configurations	JSON configuration files or API calls	Defines expected request/response patterns
Response Generator	Creates realistic provider responses	Template-based with parameter substitution	Generates valid payment provider responses
Timing Simulator	Introduces realistic delays and timeouts	Configurable delay ranges and timeout probabilities	Simulates network latency and provider processing time
State Tracker	Maintains mock payment state across requests	In-memory state store with persistence option	Tracks payment lifecycle for multi-step flows
Error Injector	Simulates various failure modes	Probability-based error injection rules	Creates realistic failure scenarios

Mock Provider Interface Design:

The mock provider implements the same interface as real payment providers, ensuring that production code remains unchanged during testing while providing extensive configurability for test scenarios.

Mock API Endpoint	Real Provider Equivalent	Configurable Behaviors	Test Scenario Support
<code>POST /v1/payment_intents</code>	Payment intent creation	Success/failure rates, response delays	Intent creation testing
<code>POST /v1/payment_intents/{id}/confirm</code>	Payment confirmation	3DS requirements, decline reasons	Payment processing testing
<code>GET /v1/payment_intents/{id}</code>	Payment status retrieval	State transitions, timing variations	Status polling testing
<code>POST /v1/refunds</code>	Refund processing	Refund approval/decline logic	Refund workflow testing
<code>POST /v1/webhooks/test</code>	Webhook delivery simulation	Event types, delivery timing, failures	Webhook handling testing

Test Scenario Configuration

Scenario Definition Structure:

Mock provider scenarios define expected request patterns and corresponding response behaviors using a structured configuration format that supports complex multi-step payment flows.

JSON

```
{  
    "scenario_name": "successful_payment_with_3ds",  
    "description": "Payment requiring 3D Secure authentication that completes successfully",  
    "request_patterns": [  
        {  
            "method": "POST",  
            "endpoint": "/v1/payment_intents/{id}/confirm",  
            "expected_params": {"payment_method": "card_*"},  
            "response": {  
                "status": "requires_action",  
                "next_action": {  
                    "type": "use_stripe_sdk",  
                    "use_stripe_sdk": {  
                        "type": "three_d_secure_redirect",  
                        "stripe_js": "https://js.stripe.com/v3/"  
                    }  
                }  
            },  
            "delay_ms": [100, 500]  
        }  
    ],  
    "webhook_events": [  
        {  
            "event_type": "payment_intent.succeeded",  
            "delay_after_request": 2000,  
            "payload_template": "payment_success_webhook.json"  
        }  
    ]  
}
```

Dynamic Scenario Management:

Test cases can modify scenario configurations during execution to simulate changing conditions or multi-phase payment flows that require different behaviors at different stages.

Configuration Method	Use Case	Implementation	Flexibility Level
Static JSON files	Standard test scenarios	Load configuration at test startup	Medium - requires file changes
Runtime API calls	Dynamic test scenarios	HTTP API to modify mock behavior	High - programmatic control
Test framework integration	Framework-driven scenarios	Pytest fixtures and parameterization	High - declarative test definition
Environment variables	Simple behavior toggles	Environment-based configuration	Low - basic on/off switches

Realistic Error Simulation

Error Injection Patterns:

The mock provider must simulate realistic error conditions that occur in production payment processing, including network issues, provider-side failures, and edge cases that are difficult to reproduce with real providers.

Error Category	Simulation Method	Realistic Triggers	Test Coverage
Network timeouts	Configurable response delays exceeding client timeouts	Random delays, connection drops	Timeout handling, retry logic
Provider errors	HTTP 5xx responses with realistic error messages	Rate limiting, maintenance windows	Error handling, circuit breaker logic
Authentication failures	Invalid signature or credential errors	Expired keys, malformed requests	Security validation, error reporting
Rate limiting	HTTP 429 responses with retry-after headers	Request frequency thresholds	Backoff logic, request throttling
Malformed responses	Invalid JSON or missing required fields	Schema violations, partial responses	Response validation, error recovery

Probabilistic Error Injection:

Rather than deterministic errors, the mock provider supports probability-based error injection that creates realistic failure patterns similar to production environments.

YAML

```

error_injection:

  network_timeouts:

    probability: 0.02 # 2% of requests timeout

    min_delay_ms: 30000

    max_delay_ms: 60000


provider_errors:

  probability: 0.01 # 1% of requests return 500 errors

  error_codes: [500, 502, 503, 504]

  retry_after_seconds: [30, 60, 120]


rate_limiting:

  requests_per_minute: 100

  burst_limit: 10

  reset_window_seconds: 60

```

3D Secure Flow Simulation

Multi-Step Authentication Mock:

3D Secure authentication requires a complex multi-step flow involving redirects, authentication challenges, and callback handling that must be accurately simulated for comprehensive testing.

3DS Flow Stage	Mock Implementation	Test Validation	User Experience Simulation
Authentication detection	Return <code>requires_action</code> for configured card patterns	Verify challenge detection logic	Realistic issuer requirements
Challenge presentation	Generate mock authentication URLs	Test redirect URL generation	Simulated issuer authentication page
User authentication	Mock user completing authentication challenge	Validate return URL handling	Various authentication outcomes
Flow completion	Process authentication result and complete payment	Test final payment confirmation	Success/failure result handling

3DS Test Scenario Configuration:

The mock provider supports various 3D Secure scenarios including successful authentication, user abandonment, issuer errors, and network failures during the authentication flow.

```
{  
  "3ds_scenarios": {  
    "successful_authentication": {  
      "challenge_required": true,  
      "authentication_delay_ms": 1500,  
      "authentication_result": "authenticated",  
      "final_payment_status": "succeeded"  
    },  
    "user_abandonment": {  
      "challenge_required": true,  
      "authentication_timeout_ms": 300000,  
      "authentication_result": "abandoned",  
      "final_payment_status": "failed"  
    },  
    "issuer_error": {  
      "challenge_required": true,  
      "authentication_delay_ms": 800,  
      "authentication_result": "error",  
      "error_code": "authentication_failed",  
      "final_payment_status": "failed"  
    }  
  }  
}
```

Webhook Delivery Simulation

Asynchronous Event Delivery:

Webhook simulation must accurately model the asynchronous nature of real payment provider webhooks, including delivery timing, retry behavior, and out-of-order event arrival.

Webhook Aspect	Simulation Approach	Configuration Options	Testing Benefit
Delivery timing	Configurable delays after triggering events	Random delays, fixed delays, sequential timing	Tests asynchronous processing
Retry behavior	Exponential backoff on endpoint failures	Retry counts, backoff multipliers, max delays	Validates retry handling
Event ordering	Intentional out-of-order delivery	Timestamp manipulation, delivery reordering	Tests event ordering logic
Signature generation	HMAC signing with configurable keys	Multiple signing keys, invalid signatures	Validates signature verification
Payload variations	Different payload formats for same events	Schema versions, optional field variations	Tests payload parsing robustness

Webhook Test Orchestration:

Test cases can orchestrate complex webhook delivery scenarios that simulate real-world conditions including network issues, endpoint downtime, and provider-side delivery problems.

```
# Webhook test orchestration example                                     PYTHON

webhook_simulator = MockWebhookDelivery()

# Configure delivery behavior for test

webhook_simulator.configure_scenario({
    "event_type": "payment_intent.succeeded",
    "delivery_attempts": 3,
    "backoff_strategy": "exponential",
    "initial_delay_ms": 1000,
    "max_delay_ms": 30000,
    "out_of_order_probability": 0.1
})

# Trigger webhook delivery

webhook_simulator.deliver_event(payment_intent_id, event_data)

# Verify eventual delivery and processing

assert webhook_simulator.verify_delivery_success(timeout_seconds=60)
```

Mock Provider Fidelity Principle

The mock provider should be sophisticated enough to catch real integration bugs while remaining simple enough to maintain reliably. Focus on simulating the behaviors that matter most for your system's correctness rather than perfect provider fidelity.

Implementation Guidance

Testing payment systems requires a multi-layered approach combining unit tests for individual components, integration tests for external dependencies, and end-to-end tests for complete payment flows. The implementation balances thorough coverage with maintainable test code that provides clear feedback when issues arise.

Technology Recommendations

Testing Layer	Simple Option	Advanced Option	Recommended For
Unit Testing	<code>pytest</code> with basic fixtures	<code>pytest</code> with <code>pytest-mock</code> and <code>hypothesis</code> for property testing	All Python components
HTTP Testing	<code>requests-mock</code> for API calls	<code>responses</code> library with callback support	Payment provider integration
Database Testing	In-memory SQLite for tests	PostgreSQL with transaction rollback per test	Data consistency validation
Mock Services	Simple HTTP mock server	<code>WireMock</code> or custom Flask app with scenario support	External provider simulation
Load Testing	<code>pytest-benchmark</code> for performance	<code>locust</code> for concurrent payment processing	Performance validation
Security Testing	Manual security test cases	<code>bandit</code> for static analysis + manual penetration testing	Security requirement validation

Recommended Test Structure

```
payment-gateway/
  tests/
    unit/                                ← Fast, isolated component tests
      test_payment_intent.py   ← PaymentIntent model and lifecycle
      test_idempotency.py     ← Idempotency key handling
      test_charge_processing.py ← Charge creation and processing
      test_refund_logic.py    ← Refund calculation and validation
      test_webhook_processing.py ← Webhook event handling
    integration/                         ← Tests with external dependencies
      test_provider_integration.py ← Payment provider API calls
      test_database_operations.py  ← Database transaction handling
      test_webhook_delivery.py    ← End-to-end webhook flows
      test_3ds_authentication.py  ← 3D Secure authentication flows
    e2e/                                  ← Complete user journey tests
      test_payment_flows.py       ← Full payment processing scenarios
      test_refund_scenarios.py   ← Complete refund workflows
      test_dispute_handling.py   ← Dispute management flows
    fixtures/                            ← Shared test data and utilities
      payment_data.py            ← Sample payment intent and charge data
      provider_responses.py     ← Mock provider response templates
      webhook_events.py         ← Sample webhook event payloads
    mocks/                               ← Mock service implementations
      mock_provider.py           ← Payment provider mock server
      mock_database.py          ← Database test utilities
  conftest.py                           ← Pytest configuration and fixtures
```

Core Testing Infrastructure

Database Test Utilities - Complete implementation for transaction-safe database testing:

```
# tests/fixtures/database.py

import pytest

from sqlalchemy import create_engine

from sqlalchemy.orm import sessionmaker

from payment_gateway.database import Base

import tempfile

import os

@pytest.fixture(scope="function")

def test_db():

    """Provides clean database session for each test with automatic rollback."""

    # Create temporary database file for isolation

    db_fd, db_path = tempfile.mkstemp()

    database_url = f"sqlite:///{{db_path}}"

    # Create engine and tables

    engine = create_engine(database_url, echo=False)

    Base.metadata.create_all(engine)

    # Create session factory

    SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

    session = SessionLocal()

    try:

        yield session

    finally:

        session.close()

        os.close(db_fd)

        os.unlink(db_path)

    @pytest.fixture
```

```
def sample_payment_intent(test_db):

    """Creates a sample payment intent for testing."""

    from payment_gateway.models import PaymentIntent, PaymentIntentStatus

    from datetime import datetime, timedelta


    intent = PaymentIntent(
        id="pi_test_123456789",
        amount=1000, # $10.00 in cents
        currency="usd",
        status=PaymentIntentStatus.created,
        idempotency_key="test_key_123",
        metadata={"customer_id": "cust_123", "order_id": "order_456"},
        created_at=datetime.utcnow(),
        expires_at=datetime.utcnow() + timedelta(hours=1),
        client_secret="pi_test_123456789_secret_abc123"
    )

    test_db.add(intent)
    test_db.commit()
    test_db.refresh(intent)

    return intent
```

Mock Payment Provider - Complete implementation with scenario support:

```
# tests/mocks/payment_provider.py

from flask import Flask, request, jsonify
import threading
import time
from typing import Dict, Any, Optional
import hmac
import hashlib
import json
from datetime import datetime, timedelta

class MockPaymentProvider:

    """Mock payment provider with configurable scenarios and realistic behaviors."""

    def __init__(self, port: int = 5555):
        self.app = Flask(__name__)
        self.port = port
        self.scenarios: Dict[str, Dict] = {}
        self.payment_states: Dict[str, Dict] = {}
        self.webhook_config = {
            "endpoint_url": None,
            "signing_secret": "whsec_test_secret_123"
        }
        self._setup_routes()
        self._server_thread = None

    def configure_scenario(self, scenario_name: str, config: Dict[str, Any]):
        """Configure behavior scenario for testing specific flows."""
        self.scenarios[scenario_name] = config

    def start_server(self):
```

```
"""Start mock provider server in background thread."""

def run_server():

    self.app.run(host='127.0.0.1', port=self.port, debug=False)

    self._server_thread = threading.Thread(target=run_server, daemon=True)
    self._server_thread.start()

    time.sleep(0.5) # Allow server to start


def _setup_routes(self):

    """Configure Flask routes for payment provider API."""

    @self.app.route('/v1/payment_intents', methods=['POST'])

    def create_payment_intent():

        # TODO: Extract request data and validate required fields

        # TODO: Generate unique payment intent ID with prefix 'pi_'

        # TODO: Check for configured scenario behaviors (delays, errors)

        # TODO: Store payment intent state for subsequent requests

        # TODO: Return realistic payment intent response with required fields

        pass

    @self.app.route('/v1/payment_intents/<intent_id>/confirm', methods=['POST'])

    def confirm_payment_intent(intent_id: str):

        # TODO: Retrieve payment intent from stored state

        # TODO: Check if 3D Secure authentication required based on scenario

        # TODO: If 3DS required, return requires_action status with challenge URL

        # TODO: If no 3DS needed, process payment and return success/failure

        # TODO: Schedule webhook delivery for payment status change

        pass

    @self.app.route('/v1/refunds', methods=['POST'])
```

```

def create_refund():

    # TODO: Validate refund request against original charge

    # TODO: Check refund amount doesn't exceed charge amount

    # TODO: Apply scenario-based success/failure logic

    # TODO: Create refund record with pending status

    # TODO: Schedule webhook for refund status update

    pass


def deliver_webhook(self, event_type: str, object_data: Dict, delay_ms: int = 0):

    """Deliver webhook to configured endpoint with realistic timing."""

    if not self.webhook_config["endpoint_url"]:

        return


def send_webhook():

    if delay_ms > 0:

        time.sleep(delay_ms / 1000.0)


    # TODO: Create webhook payload with event type and data

    # TODO: Generate HMAC signature using signing secret

    # TODO: Send POST request to webhook endpoint with signature header

    # TODO: Handle delivery failures and implement retry logic

    # TODO: Log webhook delivery attempts and outcomes

    pass


threading.Thread(target=send_webhook, daemon=True).start()

```

Payment Flow Test Scenarios - Core skeleton for comprehensive flow testing:

```
# tests/integration/test_payment_flows.py

import pytest

from payment_gateway.api import create_payment_intent, process_payment_intent

from payment_gateway.models import PaymentIntentStatus, ChargeStatus

from tests.mocks.payment_provider import MockPaymentProvider


class TestCompletePaymentFlows:

    """Integration tests for complete payment processing flows."""

    @pytest.fixture(autouse=True)

    def setup_mock_provider(self):

        """Start mock payment provider for each test."""

        self.mock_provider = MockPaymentProvider(port=5555)

        self.mock_provider.start_server()

        # Configure webhook endpoint

        self.mock_provider.webhook_config["endpoint_url"] = "http://localhost:8000/webhooks"


    def test_successful_payment_without_3ds(self, test_db):

        """Test complete payment flow for card not requiring 3D Secure."""

        # Configure mock provider for successful payment

        self.mock_provider.configure_scenario("simple_success", {

            "payment_confirmation": {

                "requires_3ds": False,

                "success": True,

                "processing_delay_ms": 500

            },

            "webhook_delivery": {

                "events": ["payment_intent.succeeded"],

                "delay_ms": 1000

            }

        })
```

```

        }

    })

# TODO: Create payment intent with test amount and currency

# TODO: Process payment intent with mock card token

# TODO: Verify charge created with succeeded status

# TODO: Wait for webhook delivery and verify local state update

# TODO: Confirm final payment intent status is succeeded


def test_successful_payment_with_3ds(self, test_db):
    """Test complete payment flow requiring 3D Secure authentication."""
    self.mock_provider.configure_scenario("3ds_success", {
        "payment_confirmation": {
            "requires_3ds": True,
            "authentication_delay_ms": 2000,
            "authentication_success": True
        }
    })

# TODO: Create payment intent and initiate processing

# TODO: Verify payment enters requires_action status

# TODO: Simulate user completing 3DS authentication

# TODO: Process authentication result and verify payment succeeds

# TODO: Validate all state transitions occurred correctly


def test_payment_failure_scenarios(self, test_db):
    """Test various payment failure conditions and error handling."""

    failure_scenarios = [
        ("insufficient_funds", "card_declined", ChargeStatus.failed),
        ("invalid_card", "invalid_request_error", ChargeStatus.failed),
    ]

```

```
("provider_timeout", "timeout_error", ChargeStatus.failed)

]

for scenario_name, error_type, expected_status in failure_scenarios:
    # TODO: Configure mock provider for specific failure scenario
    # TODO: Process payment and verify appropriate error handling
    # TODO: Confirm charge status matches expected failure state
    # TODO: Validate error details are properly recorded and surfaced
```

Milestone Testing Checkpoints

Milestone 1 Validation Script:

```
#!/bin/bash                                         BASH

# scripts/validate_milestone1.sh

echo "== Milestone 1: Payment Intent & Idempotency Validation =="

# Start test database and application server

python -m pytest tests/unit/test_payment_intent.py -v

if [ $? -ne 0 ]; then

    echo "✖ Payment intent unit tests failed"

    exit 1

fi

# Test idempotency key handling

python -m pytest tests/unit/test_idempotency.py -v

if [ $? -ne 0 ]; then

    echo "✖ Idempotency tests failed"

    exit 1

fi

# Integration test with database

python -m pytest tests/integration/test_payment_intent_lifecycle.py -v

if [ $? -ne 0 ]; then

    echo "✖ Payment intent lifecycle tests failed"

    exit 1

fi

echo "✓ Milestone 1 validation completed successfully"

echo ""

echo "Manual verification steps:"

echo "1. Start application server: python -m payment_gateway.main"

echo "2. Create payment intent: curl -X POST localhost:8000/payment-intents -d '{\"amount\": 1000, \"currency\": \"usd\"}'"

echo "3. Verify response contains 'id', 'client_secret', and status 'created'"
```

```
echo "4. Repeat request with same idempotency key, verify identical response"
```

Debugging Common Test Failures:

Test Failure Symptom	Likely Cause	Diagnostic Steps	Resolution
Payment intent creation returns 500 error	Database connection or validation error	Check database logs, verify table schema	Fix database connection string or run migrations
Idempotency tests fail intermittently	Race condition in concurrent test execution	Run tests with <code>-v</code> flag, check for timing issues	Add proper test isolation or synchronization
Mock provider tests timeout	Mock server not starting or wrong port	Check mock server logs, verify port availability	Ensure mock server starts before tests run
Webhook signature verification fails	Incorrect HMAC calculation or key mismatch	Log expected vs actual signatures, verify secret	Check webhook secret configuration and HMAC algorithm
State transition tests fail	Invalid state machine logic	Examine state transition logs and current states	Review state machine implementation and validation rules

Testing Success Metrics

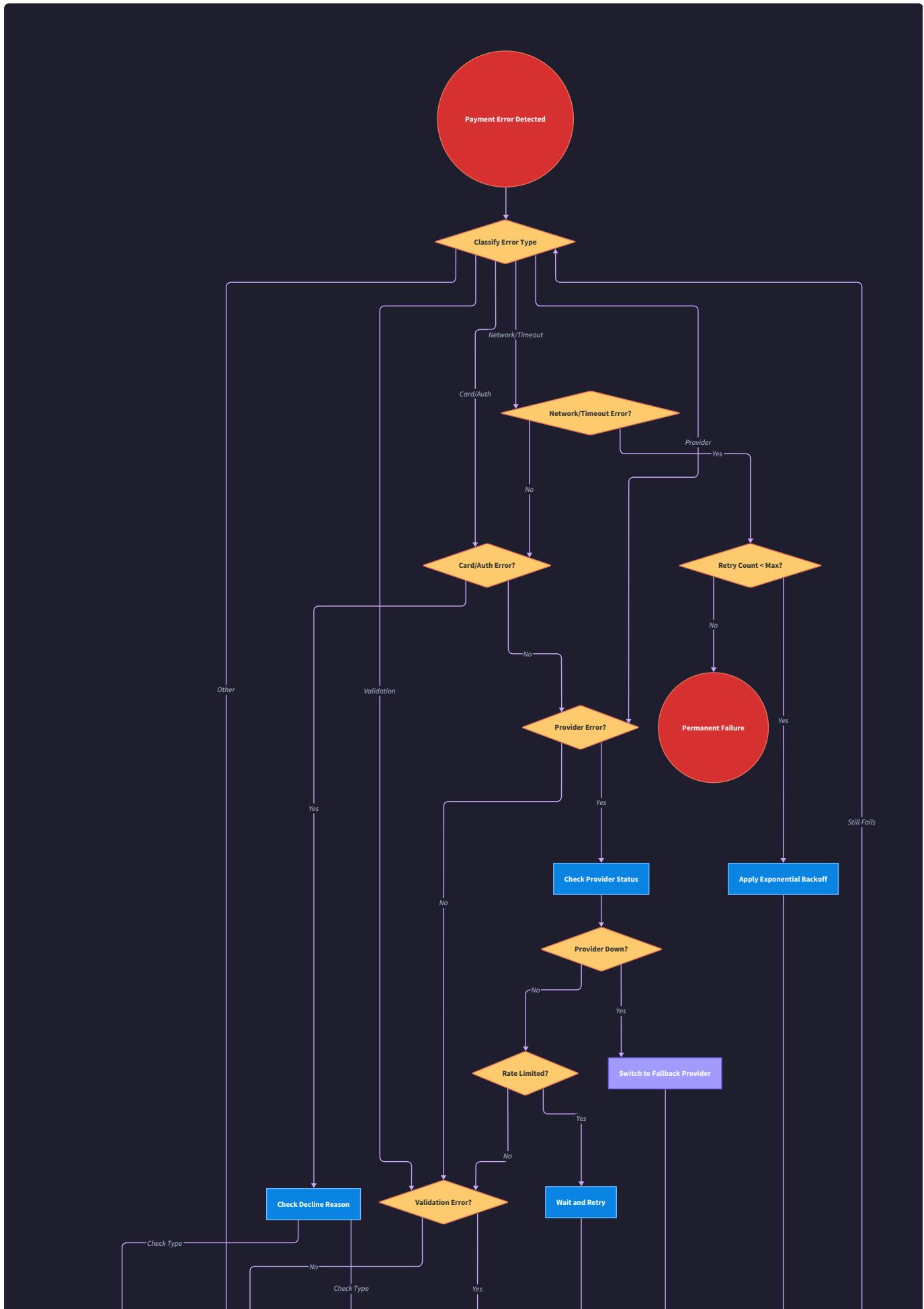
Successful milestone completion should achieve: 95%+ code coverage on core payment logic, all integration tests passing consistently, manual payment flows working end-to-end, and zero critical security test failures. Performance tests should complete within acceptable timeouts under simulated load conditions.

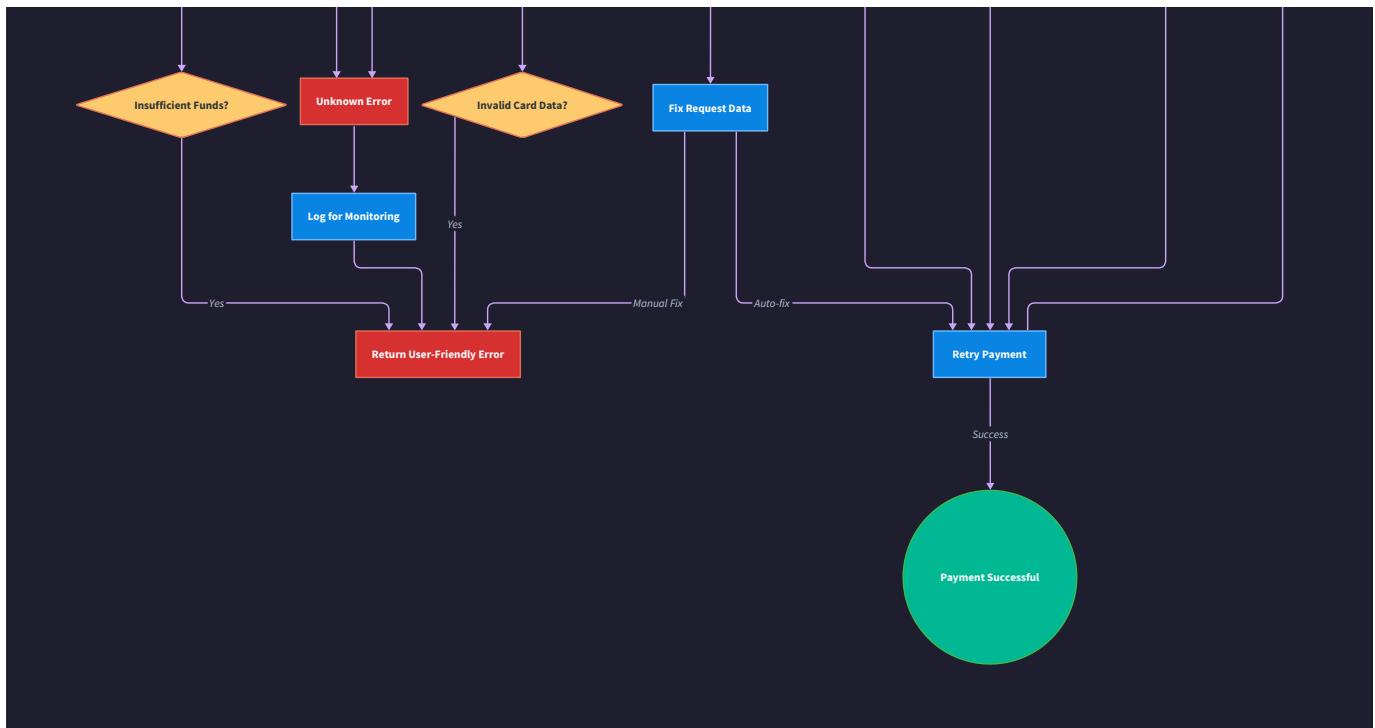
Debugging Guide

Milestone(s): All milestones - debugging skills essential for payment intent & idempotency (Milestone 1), payment processing & 3DS (Milestone 2), refunds & disputes (Milestone 3), and webhook reconciliation (Milestone 4)

Building a reliable payment processing system requires mastering the art of debugging complex distributed interactions. Think of debugging a payment gateway like being a detective investigating a crime scene - you need to collect evidence from multiple sources, understand the timeline of events, and piece together what happened when things go wrong. Unlike simple web applications where errors are often isolated to a single component, payment systems involve intricate dances between your application, external payment providers, card networks, and issuing banks. A single payment can touch dozens of systems, each with their own failure modes and timing constraints.

The complexity multiplies when you consider that payment processing is inherently asynchronous - a charge might appear to succeed locally but fail at the network level hours later, or a webhook might arrive out of order, causing your local state to diverge from reality. This section provides a systematic approach to diagnosing and resolving the most common issues encountered when building production payment systems.





Common Payment Processing Bugs

Payment processing bugs typically fall into three categories: idempotency violations, state management inconsistencies, and provider integration failures. Each category has distinct symptoms and requires different debugging approaches.

Idempotency Implementation Issues

Idempotency bugs are among the most dangerous in payment systems because they can lead to duplicate charges - the cardinal sin of payment processing. These issues often manifest subtly and may not be detected until customers report unexpected charges.

Mental Model: Think of idempotency like a notary's seal on a legal document. Once a document is notarized with a specific seal number, using that same seal number again should either return the exact same document or reject the request entirely. You can't create a different document with the same seal - that would be forgery.

The most common idempotency pitfall occurs when developers focus only on preventing duplicate database inserts but fail to validate that the new request parameters match the original request. This creates a dangerous scenario where a client can reuse an idempotency key with different payment parameters and receive the response from the original request, leading to confusion about which payment was actually processed.

Symptom	Root Cause	Diagnostic Steps	Resolution
Same idempotency key returns success but amount differs	Request parameter validation missing	Check <code>IdempotencyKey.request_hash</code> against current request	Add parameter hash validation before returning cached response
Duplicate charges appearing in provider dashboard	Race condition in idempotency check	Search logs for concurrent requests with same key	Add unique constraint on <code>idempotency_key</code> field
Client receives 500 error but payment succeeds	Exception after payment creation but before response	Check if <code>PaymentIntent</code> exists for the idempotency key	Return existing payment intent instead of creating new one
Idempotency key reuse fails after 24 hours	Idempotency record cleanup too aggressive	Verify <code>IdempotencyKey.expires_at</code> timestamp	Extend expiration or add grace period for completed payments

⚠ Pitfall: Insufficient Request Fingerprinting Many developers hash only the amount and currency when creating the request fingerprint, but miss crucial parameters like `payment_method_token` or `return_url`. If a client reuses an idempotency key with a different payment method, they'll receive the response from the original payment but the actual charge will use the original payment method, not the one in the current request.

The request hash should include all parameters that affect payment processing outcome:

```
def generate_request_hash(amount, currency, payment_method_token, metadata, return_url):    PYTHON

    """Generate deterministic hash of all parameters affecting payment outcome"""

    # Include ALL parameters that change payment behavior

    hash_input = f"{amount}:{currency}:{payment_method_token}:{json.dumps(metadata, sort_keys=True)}:{return_url}"

    return hashlib.sha256(hash_input.encode()).hexdigest()
```

⚠ Pitfall: Race Conditions During High Traffic Under concurrent load, multiple threads might simultaneously check for an existing idempotency key, find none, and proceed to create duplicate payment intents. This happens when the idempotency check and payment creation are not wrapped in a single atomic transaction.

The solution requires using database-level uniqueness constraints combined with proper exception handling:

```
-- Add unique constraint to prevent race conditions                                SQL

ALTER TABLE idempotency_keys ADD CONSTRAINT uk_idempotency_key UNIQUE (idempotency_key);
```

State Management Consistency Bugs

Payment state management bugs arise from the complex lifecycle of payment processing, where multiple entities (`PaymentIntent`, `Charge`, `Refund`) transition through various states in response to external events. These bugs often manifest as payments stuck in intermediate states or illegal state transitions.

Mental Model: Think of payment state management like air traffic control. Each payment is like an aircraft that must move through specific waypoints (states) in a predetermined sequence. Air traffic controllers (your state machine) must ensure planes don't skip waypoints or attempt impossible maneuvers. Just as two planes can't occupy the same airspace, a payment can't be in multiple contradictory states simultaneously.

Current State	Invalid Transition	Symptom	Diagnostic Query
succeeded	→ processing	Payment appears to restart after completion	SELECT * FROM charges WHERE status = 'processing' AND updated_at < NOW() - INTERVAL '1 hour'
requires_action	→ succeeded	Payment bypasses 3DS authentication	SELECT * FROM payment_intents WHERE status = 'succeeded' AND NOT EXISTS (SELECT 1 FROM charges WHERE payment_intent_id = payment_intents.id)
canceled	→ any active state	Canceled payment becomes active	SELECT * FROM payment_intents WHERE status != 'canceled' AND expires_at < NOW()
pending	→ succeeded without provider confirmation	Charge marked successful before provider webhook	SELECT * FROM charges WHERE status = 'succeeded' AND network_transaction_id IS NULL

⚠ Pitfall: Webhook-Driven State Transitions Without Validation A common mistake is allowing webhooks to transition payment states without validating that the transition is legal. For example, receiving a charge.succeeded webhook for a payment that's still in created status should raise an alert - successful charges must pass through processing status first.

Implement state transition validation in webhook handlers:

```
def can_transition_to(current_status: PaymentIntentStatus, new_status: PaymentIntentStatus) -> PYTHON
    """
    Validate legal state transitions for payment intents
    """

    valid_transitions = {

        PaymentIntentStatus.CREATED: [PaymentIntentStatus.REQUIRES_ACTION,
                                      PaymentIntentStatus.PROCESSING, PaymentIntentStatus.CANCELED],

        PaymentIntentStatus.REQUIRES_ACTION: [PaymentIntentStatus.PROCESSING,
                                              PaymentIntentStatus.CANCELED],

        PaymentIntentStatus.PROCESSING: [PaymentIntentStatus.SUCCEEDED, PaymentIntentStatus.CANCELED],

        PaymentIntentStatus.SUCCEEDED: [], # Terminal state

        PaymentIntentStatus.CANCELED: [] # Terminal state
    }

    return new_status in valid_transitions.get(current_status, [])
```

⚠ Pitfall: Ignoring Payment Intent Expiration Payment intents have limited lifespans (typically 24 hours), but many implementations forget to handle expiration properly. Expired payment intents should automatically transition to `canceled` status, but if this cleanup doesn't happen, you'll accumulate stale payments that confuse reconciliation processes.

Provider Integration Failures

Integration failures with external payment providers are inevitable in production systems. Networks fail, providers experience outages, and API rate limits are exceeded. The key is distinguishing between transient failures that warrant retries and permanent failures that require different handling.

Mental Model: Think of provider integration like international shipping. Sometimes packages get delayed due to weather (transient failure) - you wait and they eventually arrive. Sometimes packages get lost or damaged (permanent failure) - you need to file insurance claims and send replacements. The trick is knowing which situation you're in without opening every package to check.

Error Code Pattern	Failure Type	Retry Strategy	Resolution Actions
<code>502</code> , <code>503</code> , <code>504</code> HTTP errors	Provider infrastructure issue	Exponential backoff up to 5 attempts	Check provider status page, enable circuit breaker
<code>429 Too Many Requests</code>	Rate limiting	Linear backoff with jitter	Implement request queuing, review rate limits
<code>400 Invalid Request</code>	Client integration bug	No retry	Log full request/response, fix parameter validation
<code>401 Unauthorized</code>	API credential issue	No retry	Rotate API keys, check credential expiration
Network timeout	Connectivity issue	Exponential backoff up to 3 attempts	Check network connectivity, verify DNS resolution

⚠ Pitfall: Retrying Non-Idempotent Operations Not all payment provider operations are safe to retry. Creating a new charge is typically idempotent (using provider-level idempotency keys), but operations like refund creation might not be. Retrying a refund request could result in duplicate refunds if the original request succeeded but the response was lost due to network issues.

Always check provider documentation for idempotency guarantees:

```

# Safe to retry - includes idempotency key

stripe.PaymentIntent.create(
    amount=1000,
    currency='usd',
    idempotency_key=f"pi_{payment_intent.id}" # Provider-level idempotency
)

# Potentially unsafe to retry without additional checks

stripe.Refund.create(charge=charge_id, amount=500) # Check if refund already exists

```

PYTHON

⚠ Pitfall: Insufficient Error Context Logging When provider API calls fail, logging only the error code and message isn't sufficient for debugging. You need the complete context: request parameters, response headers, timing information, and correlation IDs.

Webhook Debugging Techniques

Webhook debugging requires a different mindset than traditional API debugging because you're dealing with asynchronous, push-based communication where you have limited control over timing and delivery guarantees. Webhook failures often compound over time, creating cascading inconsistencies that become harder to resolve as they age.

Mental Model: Think of webhook debugging like investigating a postal delivery failure. You need to verify that the sender actually sent the package (check provider logs), that it arrived at your address (check your server logs), that the signature matches the sender (verify HMAC), and that you processed it correctly (check your application state). If any step fails, you need different recovery strategies.

Webhook Delivery Verification

The first step in webhook debugging is confirming that webhooks are actually being sent by the provider and received by your system. Many apparent webhook failures are actually network or infrastructure issues.

Verification Step	Where to Check	What to Look For	Common Issues
Provider sent webhook	Provider dashboard/logs	Webhook delivery attempts, HTTP response codes	Provider not configured for your endpoint
Webhook reached your server	Load balancer/proxy logs	HTTP POST requests to webhook endpoint	Firewall blocking, DNS misconfiguration
Application received webhook	Application access logs	Requests with proper User-Agent, content-type	Application crashed, endpoint not mounted
Signature verification passed	Application error logs	HMAC validation errors	Wrong signing secret, clock skew issues

⚠ Pitfall: Clock Skew Affecting Signature Verification Webhook signatures often include timestamps to prevent replay attacks, but clock skew between your server and the provider's servers can cause valid webhooks to be rejected. Most

providers allow a tolerance window (typically 5-10 minutes), but this can be problematic in containerized environments where system clocks drift.

```
def verify_signature(payload, signature_header, tolerance_seconds=300):  
    """Verify webhook signature with clock skew tolerance"""  
  
    current_time = int(time.time())  
  
    # Parse timestamp from signature header  
  
    timestamp = extract_timestamp_from_signature(signature_header)  
  
    # Check if timestamp is within tolerance window  
  
    if abs(current_time - timestamp) > tolerance_seconds:  
  
        logger.warning(f"Webhook timestamp {timestamp} outside tolerance window")  
  
        # Don't immediately reject - log for monitoring but allow processing  
  
    # Continue with HMAC verification...
```

Signature Verification Issues

HMAC signature verification is critical for webhook security, but it's also a common source of integration bugs. These issues often appear intermittently, making them particularly challenging to debug.

Mental Model: Think of HMAC verification like checking a tamper-evident seal on a medicine bottle. The seal uses a special pattern that only the manufacturer knows how to create, but anyone can verify it's authentic. If the seal is broken, missing, or doesn't match the expected pattern, you know something's wrong - but you need to figure out if the problem is with the seal creation, delivery, or your verification process.

Verification Failure	Likely Cause	Debugging Steps	Resolution
All webhooks fail signature check	Wrong signing secret	Compare secret with provider dashboard	Update webhook signing secret
Intermittent signature failures	Payload modification in transit	Log raw payload bytes, check middleware	Disable payload modification middleware
Signature valid but payload corrupted	Character encoding issues	Check Content-Type header, log hex dump	Ensure UTF-8 handling throughout stack
Signature fails only for specific events	Event-specific payload format	Compare working vs failing event structures	Update signature computation for edge cases

⚠ Pitfall: Middleware Modifying Webhook Payloads Many web frameworks include middleware that automatically processes request bodies - decompressing gzip content, normalizing line endings, or parsing JSON. This processing can

alter the raw bytes used for signature computation, causing verification failures even when the webhook is legitimate.

The solution is to capture the raw request body before any middleware processing:

```
@app.before_request  
  
def capture_raw_webhook_body():  
  
    """Capture raw request body for signature verification"""  
  
    if request.path == '/webhooks/payment':  
  
        # Store raw bytes before any framework processing  
  
        g.raw_body = request.get_data()  
  
  
def verify_webhook_signature(signature_header):  
  
    """Verify signature using unmodified request body"""  
  
    raw_payload = g.raw_body # Use captured raw bytes, not request.json  
  
    return hmac.compare_digest(  
  
        expected_signature,  
  
        compute_signature(raw_payload, webhook_secret)  
  
)
```

PYTHON

Event Processing Failures

Even after successful delivery and signature verification, webhook event processing can fail due to application logic errors, database constraints, or race conditions between multiple events.

Mental Model: Think of event processing like a factory assembly line. Raw materials (webhook events) arrive at your facility and must be processed through various stations (validation, deduplication, state updates) to produce finished goods (updated payment state). If any station breaks down or processes materials in the wrong order, the entire production line backs up.

Processing Stage	Failure Symptoms	Diagnostic Queries	Recovery Actions
Event deduplication	Duplicate processing effects	<pre>SELECT COUNT(*) FROM webhook_events WHERE deduplication_key = ?</pre>	Strengthen deduplication logic, check for race conditions
Event ordering	State inconsistencies	<pre>SELECT * FROM webhook_events WHERE payment_intent_id = ? ORDER BY created_at</pre>	Implement event sequencing, reject out-of-order events
Database constraints	Constraint violation errors	Check foreign key and unique constraint violations	Implement proper error handling, add missing validation
Concurrent processing	Lost updates, race conditions	Look for simultaneous webhook processing attempts	Add row-level locking, implement event serialization

⚠ Pitfall: Processing Events Out of Order Payment providers don't guarantee webhook delivery order, so events can arrive out of sequence. For example, you might receive a `charge.succeeded` event before the corresponding `payment_intent.processing` event. Processing these out of order can leave your system in an inconsistent state.

Implement event sequencing based on timestamps:

```
def process_webhook_event(webhook_event):
    """Process webhook events in chronological order"""

    payment_intent_id = webhook_event.payload['data']['object']['id']

    # Check for newer events already processed

    latest_processed_event = db.query(WebhookEvent).filter(
        WebhookEvent.payment_intent_id == payment_intent_id,
        WebhookEvent.processed_at.is_not(None)
    ).order_by(WebhookEvent.created_at.desc()).first()

    if latest_processed_event and latest_processed_event.created_at > webhook_event.created_at:
        logger.info(f"Skipping out-of-order event {webhook_event.id}")

    return

    # Process event...
```

PYTHON

State Inconsistency Diagnosis

State inconsistencies between your local payment records and the payment provider's records are among the most serious issues in payment processing. These discrepancies can lead to accounting errors, compliance violations, and customer disputes. The key to resolving them is systematic detection, classification, and remediation.

Mental Model: Think of state reconciliation like balancing a checkbook against bank statements. Your local records are like your checkbook register - they reflect what you think happened based on the transactions you recorded. The provider's records are like the bank statement - the authoritative source of truth about what actually occurred. When they don't match, you need to investigate each discrepancy to understand why and correct your records.

Systematic State Comparison

Effective reconciliation requires comparing multiple dimensions of payment state: amounts, status values, timestamps, and reference identifiers. Each type of discrepancy indicates different root causes and requires different resolution strategies.

Discrepancy Type	Detection Method	Common Causes	Resolution Priority
Status mismatch	Compare <code>PaymentIntent.status</code> with provider status	Missed webhooks, processing delays	High - affects customer experience
Amount discrepancy	Compare <code>Charge.amount</code> with provider charge amount	Currency conversion, fee handling	Critical - affects financial reconciliation
Missing local record	Provider charge exists but no local <code>Charge</code> record	Webhook processing failure, system downtime	High - revenue recognition impact
Missing provider record	Local <code>Charge</code> exists but provider has no record	Failed API call, test vs production mixup	Critical - potential duplicate processing
Timestamp drift	Compare processing timestamps	Clock skew, timezone differences	Medium - affects reporting accuracy

The reconciliation process should run at multiple intervals: real-time validation during webhook processing, hourly checks for recent transactions, and comprehensive daily reconciliation for the previous 7 days.

```
class ReconciliationRun:

    """Represents a reconciliation execution with its results"""

    def __init__(self, run_type: str, lookback_days: int):

        self.id = str(uuid.uuid4())

        self.run_type = run_type # 'realtime', 'hourly', 'daily'

        self.lookback_days = lookback_days

        self.started_at = datetime.utcnow()

        self.processed_count = 0

        self.discrepancy_count = 0

        self.error_count = 0

        self.status = 'running'
```

⚠ Pitfall: Reconciliation During Provider Maintenance Windows Payment providers occasionally experience service disruptions or perform maintenance that affects their API responses. Running reconciliation during these windows can generate false discrepancies and trigger unnecessary alerts. Always check provider status pages and implement exponential backoff when API calls fail.

Discrepancy Classification and Auto-Resolution

Not all discrepancies require manual intervention. Many can be automatically resolved based on the type of mismatch and the age of the records involved. However, automatic resolution must be conservative - it's better to flag a discrepancy for manual review than to automatically apply an incorrect fix.

Mental Model: Think of discrepancy resolution like medical triage. Some issues (like a missing webhook that just arrived late) are minor and can be treated immediately. Others (like a significant amount discrepancy) are serious and need specialist attention. The triage system routes each case to the appropriate treatment path based on severity and confidence in the diagnosis.

Discrepancy Pattern	Auto-Resolution Conditions	Resolution Action	Manual Review Required
Local status behind provider	Age < 1 hour, valid state transition	Update local status to match provider	No
Missing webhook event	Provider event exists, local record missing	Replay event processing	No
Amount mismatch < 1%	Likely currency conversion/rounding	Update local amount, log adjustment	Yes - for audit trail
Provider refund not in local DB	Refund amount ≤ original charge	Create local refund record	Yes - verify legitimacy
Status conflict (both non-terminal)	Provider timestamp > local timestamp	Defer to provider state	Yes - investigate cause

```
def _resolve_discrepancy(discrepancy: ReconciliationDiscrepancy, db: Session):
```

PYTHON

```
    """Automatically resolve discrepancies when safe to do so"""

    if discrepancy.field_name == 'status':

        local_status = discrepancy.local_value['status']

        provider_status = discrepancy.provider_value['status']

        # Safe to auto-resolve if it's a valid forward transition
        if can_transition_to(local_status, provider_status):

            payment_intent = db.query(PaymentIntent).get(discrepancy.payment_intent_id)

            payment_intent.status = provider_status

            payment_intent.updated_at = datetime.utcnow()

            discrepancy.resolution_action = f"Updated status from {local_status} to {provider_status}"

            discrepancy.resolved_at = datetime.utcnow()

        else:

            # Backward or invalid transition requires manual review
            discrepancy.resolution_action = "Manual review required - invalid state transition"
```

⚠️ Pitfall: Over-Aggressive Auto-Resolution While automation speeds up reconciliation, being too aggressive can mask serious integration issues. For example, automatically updating local charge amounts to match provider values might hide a bug in your payment processing logic that's causing systematic amount calculation errors.

Always implement conservative thresholds and comprehensive logging for auto-resolution actions:

```
# Conservative auto-resolution thresholds  
  
AUTO_RESOLVE_AMOUNT_THRESHOLD = 0.01 # 1% or less  
  
AUTO_RESOLVE_AGE_LIMIT_HOURS = 2      # Only recent discrepancies  
  
AUTO_RESOLVE_MAX_DAILY = 100          # Circuit breaker for mass discrepancies
```

PYTHON

Root Cause Analysis for Persistent Issues

When discrepancies occur repeatedly or in patterns, they often indicate systematic issues that require deeper investigation. Root cause analysis for payment discrepancies involves examining trends, correlating with external events, and identifying common factors across failures.

Mental Model: Think of root cause analysis like epidemiological investigation during a disease outbreak. Individual cases (discrepancies) might seem random, but patterns emerge when you look at timing, affected populations (payment methods, amounts, geographic regions), and environmental factors (system deployments, provider changes, network issues).

Common root cause patterns include:

1. **Webhook Delivery Failures:** Systematic issues with webhook processing often manifest as status discrepancies that follow patterns - perhaps all payments using a specific payment method, or all payments during specific time windows when your webhook processing was experiencing issues.
2. **Integration Configuration Drift:** Changes to provider API configurations, webhook endpoints, or API credentials can cause subtle discrepancies that accumulate over time. These often correlate with deployment timestamps or configuration changes.
3. **Race Condition Windows:** Discrepancies that occur primarily during high-traffic periods might indicate race conditions in your payment processing logic, where concurrent operations interfere with each other.
4. **Provider-Side Issues:** Sometimes discrepancies cluster around provider maintenance windows, API version updates, or regional outages that affected only certain types of transactions.

Investigation Dimension	Query Pattern	What to Look For
Temporal clustering	Group discrepancies by hour/day	Spikes correlating with deployments or incidents
Payment method patterns	Group by payment method type	Specific card types or payment methods over-represented
Amount-based patterns	Group by amount ranges	Certain amounts ranges affected disproportionately
Geographic clustering	Group by customer region	Regional payment processing differences
Status transition patterns	Group by status transition type	Specific state transitions frequently inconsistent

The root cause analysis should feed back into system improvements - updating monitoring alerts, strengthening validation logic, or implementing additional safeguards to prevent similar issues in the future.

Implementation Guidance

Building effective debugging capabilities requires proactive instrumentation, structured logging, and automated monitoring that can detect issues before they impact customers.

Technology Recommendations

Component	Simple Option	Advanced Option
Logging Framework	Python <code>logging</code> with JSON formatter	Structured logging with <code>structlog</code>
Error Tracking	File-based error logs	Sentry or Rollbar for error aggregation
Metrics Collection	Prometheus with custom counters	DataDog or New Relic APM
Webhook Testing	<code>ngrok</code> for local development	Webhook.site for payload inspection
State Comparison	Manual SQL queries	Custom reconciliation dashboard
Log Analysis	<code>grep</code> and command-line tools	ELK stack or Splunk for log search

Debugging Infrastructure Setup

Create comprehensive debugging infrastructure that captures all necessary context for payment processing issues:

```
# debug_infrastructure.py

import logging

import json

import time

from datetime import datetime

from typing import Dict, Any, Optional

from contextlib import contextmanager

from functools import wraps


class PaymentDebugLogger:

    """Centralized logging for payment processing with structured context"""

    def __init__(self):

        self.logger = logging.getLogger('payment_gateway')

        self.logger.setLevel(logging.INFO)

        # JSON formatter for structured logging

        handler = logging.StreamHandler()

        formatter = logging.Formatter(

            '%(asctime)s %(levelname)s %(name)s %(message)s'

        )

        handler.setFormatter(formatter)

        self.logger.addHandler(handler)

    def log_payment_event(self, event_type: str, payment_intent_id: str,

                           context: Dict[str, Any], level: str = 'info'):

        """Log payment events with standardized context"""

        log_data = {

            'event_type': event_type,

            'payment_intent_id': payment_intent_id,
```

```
'timestamp': datetime.utcnow().isoformat(),

'context': context

}

getattr(self.logger, level)(json.dumps(log_data))



@contextmanager

def payment_operation_context(self, operation: str, payment_intent_id: str):

    """Context manager for tracking payment operations"""

    start_time = time.time()

    operation_id = f"{operation}_{int(start_time)}"

    self.log_payment_event('operation_started', payment_intent_id, {

        'operation': operation,

        'operation_id': operation_id

    })



try:

    yield operation_id

    duration = time.time() - start_time

    self.log_payment_event('operation_completed', payment_intent_id, {

        'operation': operation,

        'operation_id': operation_id,

        'duration_seconds': duration

    })

except Exception as e:

    duration = time.time() - start_time

    self.log_payment_event('operation_failed', payment_intent_id, {

        'operation': operation,

        'operation_id': operation_id,
```

```
'duration_seconds': duration,
    'error_type': type(e).__name__,
    'error_message': str(e)
}, level='error')
raise

debug_logger = PaymentDebugLogger()

def log_payment_operation(operation_name: str):
    """Decorator for automatic payment operation logging"""

    def decorator(func):
        @wraps(func)

        def wrapper(*args, **kwargs):
            # Extract payment_intent_id from function arguments
            payment_intent_id = kwargs.get('payment_intent_id') or (
                args[0] if args and hasattr(args[0], 'id') else 'unknown'
            )

            with debug_logger.payment_operation_context(operation_name, payment_intent_id):
                return func(*args, **kwargs)

        return wrapper

    return decorator
```

Webhook Debugging Tools

Build specialized tools for webhook debugging that can capture, replay, and analyze webhook events:

PYTHON

```
# webhook_debugger.py

import hmac

import hashlib

import json

from datetime import datetime, timedelta

from typing import Optional, List, Dict

from dataclasses import dataclass


@dataclass

class WebhookDebugInfo:

    """Captured webhook debugging information"""

    raw_payload: bytes

    headers: Dict[str, str]

    signature_header: str

    timestamp: datetime

    verification_result: bool

    verification_error: Optional[str]

    processing_result: Optional[str]


class WebhookDebugger:

    """Tools for debugging webhook delivery and processing issues"""


    def __init__(self, webhook_secret: str):

        self.webhook_secret = webhook_secret

        self.captured_webhooks: List[WebhookDebugInfo] = []


    def capture_webhook(self, raw_payload: bytes, headers: Dict[str, str], signature_header: str) -> WebhookDebugInfo:

        """Capture webhook for debugging analysis"""

        verification_result, verification_error = self._verify_signature(
            raw_payload, signature_header
```

```
)  
  
    debug_info = WebhookDebugInfo(  
        raw_payload=raw_payload,  
        headers=headers,  
        signature_header=signature_header,  
        timestamp=datetime.utcnow(),  
        verification_result=verification_result,  
        verification_error=verification_error,  
        processing_result=None  
    )  
  
    self.captured_webhooks.append(debug_info)  
  
    return debug_info  
  
  
def _verify_signature(self, payload: bytes, signature_header: str) -> tuple[bool, Optional[str]]:  
    """Verify webhook signature with detailed error reporting"""  
  
    try:  
  
        # Parse signature header (format: "t=timestamp,v1=signature")  
  
        sig_parts = dict(part.split('=', 1) for part in signature_header.split(','))  
  
        timestamp = sig_parts.get('t')  
  
        signature = sig_parts.get('v1')  
  
  
        if not timestamp or not signature:  
  
            return False, "Invalid signature header format"  
  
  
        # Check timestamp age  
  
        webhook_time = datetime.fromtimestamp(int(timestamp))  
  
        age_minutes = (datetime.utcnow() - webhook_time).total_seconds() / 60
```

```
        if age_minutes > 5: # 5-minute tolerance

            return False, f"Webhook too old: {age_minutes:.1f} minutes"

    # Compute expected signature

    signed_payload = f"{timestamp}.{payload.decode('utf-8')}"

    expected_signature = hmac.new(
        self.webhook_secret.encode(),
        signed_payload.encode(),
        hashlib.sha256
    ).hexdigest()

    if not hmac.compare_digest(signature, expected_signature):

        return False, "HMAC signature mismatch"

    return True, None

except Exception as e:

    return False, f"Signature verification exception: {str(e)}"

def replay_webhook(self, debug_info: WebhookDebugInfo) -> str:

    """Replay a captured webhook for testing"""

    # TODO: Implement webhook replay logic

    # This would re-process the webhook event using the captured payload

    pass

def analyze_webhook_patterns(self, hours_back: int = 24) -> Dict[str, Any]:

    """Analyze webhook patterns for debugging insights"""

    cutoff_time = datetime.utcnow() - timedelta(hours=hours_back)

    recent_webhooks = [
        w for w in self.captured_webhooks
```

```
        if w.timestamp > cutoff_time
    ]

return {
    'total_webhooks': len(recent_webhooks),
    'verification_success_rate': sum(1 for w in recent_webhooks if w.verification_result) / len(recent_webhooks),
    'common_verification_errors': self._group_errors([w.verification_error for w in recent_webhooks if w.verification_error]),
    'event_types': self._extract_event_types(recent_webhooks),
    'processing_success_rate': sum(1 for w in recent_webhooks if w.processing_result == 'success') / len(recent_webhooks)
}
```

State Reconciliation Tools

Implement comprehensive reconciliation tools that can detect and categorize state discrepancies:

```
# reconciliation_tools.py

from datetime import datetime, timedelta

from typing import List, Dict, Any, Optional

from dataclasses import dataclass

import logging

@dataclass
class StateDiscrepancy:

    """Represents a detected state discrepancy"""

    payment_intent_id: str

    entity_type: str # 'payment_intent', 'charge', 'refund'

    field_name: str

    local_value: Any

    provider_value: Any

    severity: str # 'low', 'medium', 'high', 'critical'

    auto_resolvable: bool

    detection_timestamp: datetime

class PaymentStateReconciler:

    """Tools for reconciling payment state with provider"""

    def __init__(self, provider_client, db_session):

        self.provider = provider_client

        self.db = db_session

        self.logger = logging.getLogger(__name__)

    def reconcile_payment_intent(self, payment_intent_id: str) -> List[StateDiscrepancy]:

        """Reconcile single payment intent with provider state"""

        discrepancies = []

        # Fetch local payment intent
```

```

local_payment = self.db.query(PaymentIntent).get(payment_intent_id)

if not local_payment:

    # This is a critical issue - should not happen

    return []


try:

    # Fetch provider payment intent

    provider_payment = self.provider.get_payment_intent(payment_intent_id)


    # Compare status

    if local_payment.status != provider_payment.status:

        discrepancies.append(StateDiscrepancy(
            payment_intent_id=payment_intent_id,
            entity_type='payment_intent',
            field_name='status',
            local_value=local_payment.status,
            provider_value=provider_payment.status,
            severity=self._assess_status_discrepancy_severity(
                local_payment.status, provider_payment.status
            ),
            auto_resolvable=self._can_auto_resolve_status(
                local_payment.status, provider_payment.status
            ),
            detection_timestamp=datetime.utcnow()
        ))


    # Compare amount

    if local_payment.amount != provider_payment.amount:

        discrepancies.append(StateDiscrepancy(
            payment_intent_id=payment_intent_id,

```

```

        entity_type='payment_intent',
        field_name='amount',
        local_value=local_payment.amount,
        provider_value=provider_payment.amount,
        severity='critical', # Amount discrepancies are always critical
        auto_resolvable=False, # Never auto-resolve amount discrepancies
        detection_timestamp=datetime.utcnow()

    )))

# Reconcile associated charges

discrepancies.extend(self._reconcile_charges(payment_intent_id, provider_payment))

except Exception as e:

    self.logger.error(f"Failed to reconcile payment {payment_intent_id}: {e}")

    # Could indicate provider API issues or network problems


return discrepancies

def _assess_status_discrepancy_severity(self, local_status: str, provider_status: str) -> str:

    """Assess severity of status discrepancies"""

    # Local behind provider - common due to webhook delays

    if local_status == 'processing' and provider_status == 'succeeded':

        return 'medium'

    # Local ahead of provider - unusual and concerning

    if local_status == 'succeeded' and provider_status == 'processing':

        return 'high'

    # Terminal state mismatches are critical

    if local_status in ['succeeded', 'canceled'] and provider_status in ['succeeded', 'canceled']:

```

```
        return 'critical'

    return 'medium'

def _can_auto_resolve_status(self, local_status: str, provider_status: str) -> bool:
    """Determine if status discrepancy can be safely auto-resolved"""

    # Only allow forward transitions that are valid in our state machine
    return can_transition_to(local_status, provider_status)

def run_batch_reconciliation(self, lookback_hours: int = 24) -> Dict[str, Any]:
    """Run batch reconciliation for recent payments"""

    cutoff_time = datetime.utcnow() - timedelta(hours=lookback_hours)

    recent_payments = self.db.query(PaymentIntent).filter(
        PaymentIntent.created_at > cutoff_time
    ).all()

    all_discrepancies = []
    processing_errors = []

    for payment in recent_payments:
        try:
            discrepancies = self.reconcile_payment_intent(payment.id)
            all_discrepancies.extend(discrepancies)
        except Exception as e:
            processing_errors.append({
                'payment_intent_id': payment.id,
                'error': str(e)
            })
```

```

    return {

        'total_payments_checked': len(recent_payments),

        'total_discrepancies': len(all_discrepancies),

        'discrepancies_by_severity': self._group_by_severity(all_discrepancies),

        'auto_resolvable_count': sum(1 for d in all_discrepancies if d.auto_resolvable),

        'processing_errors': processing_errors,

        'completion_timestamp': datetime.utcnow().isoformat()

    }

```

Milestone Debugging Checkpoints

Implement specific debugging checkpoints for each milestone to help learners validate their implementation:

Milestone 1: Payment Intent & Idempotency Debugging

```

# Test idempotency key handling                                                 BASH

curl -X POST http://localhost:8000/payment-intents \
-H "Content-Type: application/json" \
-H "Idempotency-Key: test-key-001" \
-d '{"amount": 1000, "currency": "usd"}'

# Make identical request - should return same payment intent

curl -X POST http://localhost:8000/payment-intents \
-H "Content-Type: application/json" \
-H "Idempotency-Key: test-key-001" \
-d '{"amount": 1000, "currency": "usd"}'

# Check database state

psql -c "SELECT id, status, idempotency_key FROM payment_intents WHERE idempotency_key = 'test-key-001';"

psql -c "SELECT idempotency_key, request_hash, response_status FROM idempotency_keys WHERE idempotency_key = 'test-key-001';"

```

Expected Results:

- Both API calls return identical response (same payment intent ID)
- Database contains exactly one payment intent record

- Idempotency key table contains one record with successful response cached

Milestone 2: Payment Processing & 3DS Debugging

```
# Test payment method tokenization
curl -X POST http://localhost:8000/payment-methods \
-H "Content-Type: application/json" \
-d '{"type": "card", "card": {"number": "4000002760003184", "exp_month": 12, "exp_year": 2025,
"cvc": "123"}}

# Process payment intent with 3DS card
curl -X POST http://localhost:8000/payment-intents/pi_test_123/process \
-H "Content-Type: application/json" \
-d '{"payment_method_token": "pm_test_456"}'

# Check for 3DS redirect URL in response
# Verify charge status is 'pending' with requires_action
```

BASH

Common Debugging Scenarios

Scenario	Symptoms	Investigation Steps	Typical Resolution
Idempotency key collision	409 error with different request params	Check <code>request_hash</code> mismatch in logs	Fix request parameter hashing
Webhook signature failure	All webhooks rejected with 401	Verify webhook secret configuration	Update webhook signing secret
Payment stuck in processing	Status never updates after charge	Check webhook delivery in provider dashboard	Manually trigger webhook or fix endpoint
3DS redirect loop	User redirected repeatedly	Check <code>return_url</code> parameter validity	Fix return URL generation logic
Amount discrepancy failure	Local amount ≠ provider amount	Check currency conversion and fee calculation	Fix amount calculation or currency handling
Refund processing failure	Refund API calls fail	Check original charge status and refund eligibility	Verify charge is settled and refundable

Future Extensions

Milestone(s): Foundation for future development - architectural patterns established across all milestones enable scaling to multi-provider support, subscription billing, marketplace payments, and advanced fraud detection capabilities.

The current payment gateway implementation provides a solid foundation for processing individual transactions, handling refunds, and managing webhook reconciliation. However, real-world payment systems often require additional capabilities that extend far beyond basic transaction processing. This section explores the architectural patterns and design considerations needed to evolve our payment gateway into a comprehensive financial platform capable of supporting complex business models and enterprise-scale operations.

Think of our current payment system as a well-built bicycle - it handles the core transportation need effectively, but different journeys require different vehicles. A delivery service needs trucks with multiple compartments, a taxi service needs cars with passenger management systems, and a logistics company needs a fleet management system coordinating multiple vehicles. Similarly, our payment gateway needs different architectural extensions for different business requirements: multi-provider support for reliability and cost optimization, subscription billing for recurring revenue models, marketplace payments for multi-party transactions, and fraud detection for risk management.

The key insight driving these extensions is that each new capability should build upon our existing foundation rather than replacing it. The `PaymentIntent` state machine, idempotency controls, webhook reconciliation, and error handling patterns we've established provide the architectural skeleton onto which we can attach new functionality. This approach ensures consistency across features while allowing each extension to solve its specific domain problems.

Key Design Principle: Extensions should enhance the core payment processing engine without fundamentally altering its reliability guarantees or introducing architectural complexity that compromises the basic transaction flow.

Multi-Provider Architecture

Modern payment processing demands resilience, cost optimization, and geographic coverage that no single payment provider can deliver comprehensively. Think of multi-provider architecture like having multiple internet service providers for a critical data center - each provider has different strengths, coverage areas, cost structures, and reliability characteristics. When processing payments across different regions, currencies, or payment methods, routing transactions to the optimal provider while maintaining consistent behavior requires careful architectural design.

The fundamental challenge in multi-provider architecture is maintaining our existing `PaymentIntent` abstraction while introducing provider-specific behaviors, capabilities, and error conditions. Our current system directly integrates with a single provider's API, making assumptions about response formats, webhook signatures, and transaction lifecycle timing. Extending to multiple providers requires introducing an abstraction layer that normalizes these differences while preserving provider-specific optimizations.

Provider Abstraction Layer Design

The provider abstraction layer serves as the translation boundary between our domain model and provider-specific implementations. Consider how a universal remote control abstracts different television manufacturers' control protocols - it presents consistent buttons (power, volume, channel) while translating each button press into manufacturer-specific

infrared signals. Similarly, our provider abstraction translates standardized payment operations into provider-specific API calls.

The abstraction centers around a `PaymentProvider` interface that defines the essential operations every provider must support, regardless of their underlying implementation differences:

Method	Parameters	Returns	Description
<code>create_payment_intent</code>	<code>amount</code> , <code>currency</code> , <code>metadata</code> , <code>idempotency_key</code>	<code>ProviderPaymentIntent</code>	Creates payment intent with provider-specific details
<code>create_charge</code>	<code>payment_intent_id</code> , <code>payment_method_token</code> , <code>return_url</code>	<code>ProviderChargeResponse</code>	Initiates charge processing with authentication flow
<code>confirm_payment</code>	<code>provider_charge_id</code> , <code>authentication_token</code>	<code>ProviderChargeResponse</code>	Completes 3D Secure authentication
<code>create_refund</code>	<code>provider_charge_id</code> , <code>amount</code> , <code>reason</code>	<code>ProviderRefundResponse</code>	Processes full or partial refund
<code>get_payment_status</code>	<code>provider_payment_id</code>	<code>ProviderPaymentStatus</code>	Retrieves current payment state for reconciliation
<code>verify_webhook_signature</code>	<code>payload</code> , <code>signature_header</code> , <code>secret</code>	<code>bool</code>	Validates webhook authenticity using provider's method
<code>parse_webhook_event</code>	<code>payload</code> , <code>event_type</code>	<code>StandardWebhookEvent</code>	Normalizes provider webhook into standard format

Each provider implementation encapsulates the specific API client, authentication mechanisms, request/response transformations, and error code mappings needed for that provider. This encapsulation allows the core payment processing engine to remain unchanged while supporting dramatically different provider architectures.

Decision: Provider Interface Abstraction

- **Context:** Multiple payment providers have different API designs, authentication methods, and data formats, but our core payment processing logic should remain consistent
- **Options Considered:** Direct provider switching, adapter pattern, strategy pattern
- **Decision:** Strategy pattern with provider interface abstraction
- **Rationale:** Strategy pattern allows runtime provider selection while maintaining type safety and testability; adapter pattern would require creating adapters for each provider combination; direct switching would require conditional logic throughout the payment engine
- **Consequences:** Clean separation of provider-specific logic, easy testing with mock providers, but requires careful design of the abstraction to avoid lowest-common-denominator limitations

Provider Selection Strategy

Selecting the optimal provider for each transaction requires a decision engine that evaluates multiple factors including transaction characteristics, provider capabilities, cost structures, and current system health. Think of this like routing network traffic through different internet service providers - the router considers latency, bandwidth, cost, and current congestion to select the best path for each packet.

The provider selection strategy operates through a configurable rule engine that evaluates selection criteria in priority order:

Selection Criteria	Evaluation Logic	Example Rules
Geographic Routing	Match customer billing country to provider coverage	Route EU customers to European acquirer for lower interchange fees
Currency Support	Verify provider supports transaction currency with favorable rates	Use local acquiring for same-currency transactions to minimize conversion costs
Payment Method	Match payment method capabilities (cards, bank transfers, wallets)	Route Apple Pay to provider with direct Apple Pay integration
Transaction Amount	Consider provider fee structures and processing limits	Route high-value transactions to provider with lower percentage fees
Provider Health	Evaluate current success rates, latency, and circuit breaker status	Avoid providers experiencing elevated decline rates or API timeouts
Business Rules	Apply merchant-specific routing preferences or compliance requirements	Route transactions from specific MCC codes to compliant processors

The selection engine maintains real-time metrics for each provider, including success rates, average processing times, and current error rates. These metrics feed into the decision algorithm, automatically shifting traffic away from providers experiencing issues while maintaining detailed audit logs for analysis.

Provider Selection Algorithm:

1. Filter providers by hard requirements (currency support, payment method, geographic licensing)
2. Apply business rules and compliance constraints to remaining providers
3. Evaluate each qualifying provider using weighted scoring (cost 40%, reliability 30%, performance 30%)
4. Select highest-scoring provider unless circuit breaker indicates unavailability
5. Log selection decision with full scoring breakdown for analysis
6. Fall back to secondary provider if primary selection fails during transaction processing

Provider Health Monitoring

Maintaining optimal routing decisions requires continuous monitoring of provider performance, reliability, and cost effectiveness. Consider how a traffic management system monitors highway conditions - it tracks congestion levels, accident reports, construction delays, and weather conditions to provide real-time routing recommendations. Similarly, provider health monitoring tracks multiple dimensions of provider performance to inform routing decisions.

The monitoring system collects metrics across several key dimensions:

Metric Category	Key Indicators	Update Frequency	Decision Impact
Success Rates	Authorization success, settlement success, 3DS completion	Real-time per transaction	Primary routing factor - avoid providers with declining success
Performance	API response times, 3DS redirect latency, settlement timing	Per API call	Secondary factor - prefer faster providers for time-sensitive flows
Cost Analysis	Effective interchange rates, provider fees, currency conversion	Daily batch analysis	Strategic factor - optimize for total cost of processing
Compliance Status	PCI certification status, regional licensing, fraud monitoring	Weekly validation	Hard constraint - disable non-compliant providers immediately

Provider health monitoring integrates with our existing circuit breaker patterns, automatically removing unhealthy providers from the selection pool while implementing gradual recovery testing. When a provider's performance degrades below acceptable thresholds, the system reduces traffic allocation progressively rather than immediately blacklisting the provider, allowing for temporary issues to resolve without overreacting to transient problems.

Advanced Payment Features

Beyond basic transaction processing, modern payment systems support sophisticated business models that require specialized payment flows, billing cycles, and multi-party transaction management. These advanced features build upon our core payment infrastructure while introducing new domain concepts and state management requirements.

Subscription Billing Architecture

Subscription billing transforms one-time payment processing into a recurring revenue management system. Think of subscription billing like a magazine subscription service - instead of purchasing individual issues, customers commit to receiving regular deliveries with automatic payment processing, while the service manages billing cycles, prorated charges, plan changes, and renewal notifications.

The subscription billing architecture introduces new domain entities that extend our existing payment model:

Entity	Key Fields	Description
Subscription	<code>id, customer_id, plan_id, status,</code> <code>current_period_start, current_period_end,</code> <code>payment_method_token</code>	Core subscription configuration with billing cycle timing
SubscriptionPlan	<code>id, amount, currency, interval, interval_count,</code> <code>trial_period_days</code>	Billing plan template defining pricing and frequency
BillingCycle	<code>id, subscription_id, period_start, period_end,</code> <code>amount_due, status</code>	Individual billing period with payment tracking
Usage	<code>id, subscription_id, metric_name, quantity,</code> <code>billing_cycle_id</code>	Usage-based billing tracking for metered subscriptions
SubscriptionInvoice	<code>id, subscription_id, billing_cycle_id,</code> <code>line_items, total_amount, due_date</code>	Detailed billing statement with itemized charges

The subscription billing engine orchestrates complex lifecycle management including trial periods, plan upgrades, prorated billing adjustments, and failed payment recovery. Each billing cycle generates a `PaymentIntent` using our existing infrastructure, ensuring subscription payments benefit from the same idempotency controls, 3D Secure authentication, and webhook reconciliation as one-time transactions.

Decision: Subscription State Management

- **Context:** Subscriptions have complex lifecycle states (trial, active, past_due, canceled) with different billing rules and customer communication requirements
- **Options Considered:** Extend `PaymentIntent` states, separate subscription state machine, event-driven state management
- **Decision:** Separate subscription state machine with event-driven transitions
- **Rationale:** Subscription states operate on different timescales than payment states; separating concerns allows subscription logic to evolve independently while maintaining payment processing reliability
- **Consequences:** Clean separation of billing and payment concerns, easier testing and debugging, but requires careful coordination between subscription and payment state machines

Marketplace Payment Architecture

Marketplace payments enable multi-party transactions where funds flow from buyers to sellers through the marketplace platform, with the platform collecting fees and handling compliance obligations. Consider how a real estate transaction involves multiple parties - the buyer, seller, real estate agents, escrow company, and title company - each with different roles, fee structures, and payout timing requirements.

Marketplace payments introduce the concept of **connected accounts** and **transfer groups** that extend our payment model to support complex fund distribution:

Entity	Key Fields	Description
ConnectedAccount	<code>id</code> , <code>marketplace_id</code> , <code>account_holder_id</code> , <code>status</code> , <code>capabilities</code> , <code>verification_status</code>	Seller account with platform verification and capabilities
TransferGroup	<code>id</code> , <code>payment_intent_id</code> , <code>total_amount</code> , <code>platform_fee</code> , <code>status</code>	Group of related transfers from single payment
Transfer	<code>id</code> , <code>transfer_group_id</code> , <code>destination_account_id</code> , <code>amount</code> , <code>description</code> , <code>status</code>	Individual transfer to connected account
PlatformFee	<code>id</code> , <code>transfer_id</code> , <code>fee_type</code> , <code>amount</code> , <code>description</code>	Platform fees collected from transfer

The marketplace payment flow coordinates multiple operations atomically - authorizing the full payment amount from the buyer, distributing funds to sellers after deducting platform fees, handling tax reporting requirements, and managing payout schedules. This coordination requires careful orchestration to ensure all parties receive correct amounts while maintaining audit trails for financial reconciliation.

Marketplace payments also introduce complex compliance requirements including:

- **KYC (Know Your Customer) verification** for connected accounts before enabling payouts
- **Tax reporting** for seller earnings and platform fees across multiple jurisdictions
- **Funds flow transparency** allowing buyers and sellers to track payment distribution
- **Dispute management** handling conflicts between buyers, sellers, and the platform
- **Regulatory compliance** meeting marketplace-specific regulations like payment service provider licensing

Fraud Detection Integration

Advanced payment systems require sophisticated fraud detection capabilities that analyze transaction patterns, customer behavior, and external risk signals to identify potentially fraudulent activity. Think of fraud detection like an airport security system - it uses multiple screening mechanisms (ID verification, X-ray scanning, behavioral observation) to identify potential threats while minimizing disruption to legitimate travelers.

Fraud detection integration operates through a **risk scoring pipeline** that evaluates each transaction against multiple risk factors:

Risk Category	Analysis Factors	Risk Signals
Transaction Patterns	Amount relative to customer history, frequency of transactions, time-of-day patterns	Sudden large transactions, unusual transaction timing, velocity violations
Device Fingerprinting	Browser characteristics, IP address, device ID persistence	New device usage, proxy/VPN detection, device spoofing indicators
Geographic Analysis	Billing vs shipping address, IP geolocation, known high-risk countries	Address mismatches, impossible travel patterns, high-risk jurisdictions
Payment Method Validation	Card BIN analysis, payment method age, previous decline history	New payment method usage, stolen card list matches, repeated decline patterns
Customer Behavior	Account age, purchase history, social signals	New account risk, behavioral anomalies, social media verification

The fraud detection system integrates with our payment processing flow by evaluating risk scores during payment intent creation and charge processing. High-risk transactions can be automatically declined, flagged for manual review, or subjected to additional authentication requirements like step-up verification or manual approval workflows.

Decision: Real-time vs Batch Fraud Analysis

- **Context:** Fraud detection requires balancing transaction processing speed against analysis thoroughness, with different risk factors having different computational requirements
- **Options Considered:** All real-time analysis, hybrid real-time/batch analysis, all batch analysis with post-transaction review
- **Decision:** Hybrid approach with real-time basic checks and batch analysis for sophisticated modeling
- **Rationale:** Basic checks (velocity, blacklists, device fingerprinting) can be performed quickly during transaction flow; complex machine learning models and cross-account pattern analysis require batch processing
- **Consequences:** Fast transaction processing for most payments, comprehensive risk analysis without blocking customer experience, but requires careful coordination between real-time and batch systems

⚠ Pitfall: Over-aggressive Fraud Detection Many fraud detection implementations err on the side of caution, implementing strict rules that generate high false positive rates. This approach damages customer experience by declining legitimate transactions, especially for international customers or unusual purchase patterns. Design fraud detection systems with gradual response escalation - implement step-up authentication, manual review queues, and customer communication before automatically declining transactions. Monitor false positive rates as carefully as fraud detection rates to maintain optimal customer experience.

Implementation Guidance

The future extensions described above represent significant architectural expansions that require careful planning and phased implementation. Each extension builds upon our existing payment infrastructure while introducing new technical challenges around data consistency, transaction coordination, and system integration.

Technology Recommendations

Component	Simple Option	Advanced Option
Provider Abstraction	Strategy pattern with direct API clients	Abstract factory with provider plugins and configuration management
Subscription Management	Cron-based billing cycle processing	Event-driven billing with distributed job queue
Marketplace Transfers	Synchronous transfer creation during payment	Asynchronous transfer processing with eventual consistency
Fraud Detection	Rule-based scoring with external API integration	Real-time ML pipeline with feature store and model serving
Multi-tenant Architecture	Database-per-tenant with connection pooling	Shared database with row-level security and query optimization

Recommended File Structure

These extensions require significant additional code organization to maintain the clean architecture established in our core payment processing engine:

```

payment-gateway/
├── core/
│   ├── payment_intent.py      ← existing core payment logic
│   ├── charge_processing.py  ← existing charge processing
│   └── webhook_handler.py   ← existing webhook processing
├── providers/
│   ├── __init__.py
│   ├── base_provider.py     ← PaymentProvider interface definition
│   ├── stripe_provider.py   ← Stripe-specific implementation
│   ├── adyen_provider.py    ← Adyen-specific implementation
│   └── provider_factory.py  ← Provider selection and instantiation
├── subscription/
│   ├── __init__.py
│   ├── models.py            ← Subscription, Plan, BillingCycle entities
│   ├── billing_engine.py    ← Subscription billing orchestration
│   ├── plan_manager.py     ← Plan changes and prorations
│   └── trial_handler.py    ← Trial period management
├── marketplace/
│   ├── __init__.py
│   ├── models.py            ← ConnectedAccount, Transfer entities
│   ├── transfer_engine.py   ← Multi-party payment distribution
│   ├── fee_calculator.py   ← Platform fee calculation logic
│   └── compliance.py       ← KYC and tax reporting
├── fraud/
│   ├── __init__.py
│   ├── risk_scorer.py      ← Real-time risk score calculation
│   ├── rule_engine.py       ← Configurable fraud rules
│   ├── device_analyzer.py  ← Device fingerprinting logic
│   └── batch_analyzer.py   ← Batch ML model processing
└── monitoring/
    ├── provider_health.py  ← Provider performance monitoring
    ├── metrics_collector.py← System-wide metrics aggregation
    └── alerting.py          ← Automated alert generation

```

Provider Abstraction Implementation

The provider abstraction layer requires careful interface design to support provider-specific capabilities while maintaining consistency across implementations:

```
from abc import ABC, abstractmethod

from typing import Dict, Any, Optional

from dataclasses import dataclass

from enum import Enum


class ProviderCapability(Enum):

    CARD_PAYMENTS = "card_payments"

    BANK_TRANSFERS = "bank_transfers"

    DIGITAL_WALLETS = "digital_wallets"

    RECURRING_BILLING = "recurring_billing"

    MARKETPLACE_TRANSFERS = "marketplace_transfers"

    INSTANT_PAYOUTS = "instant_payouts"


@dataclass

class ProviderConfig:

    """Configuration for provider-specific settings and capabilities."""

    provider_name: str

    api_endpoint: str

    api_key: str

    webhook_secret: str

    supported_currencies: set[str]

    capabilities: set[ProviderCapability]

    fee_structure: Dict[str, float]

    processing_delay: int # seconds

    max_amount: int # cents

    min_amount: int # cents


@dataclass

class ProviderPaymentIntent:

    """Normalized payment intent response from provider."""

    provider_id: str
```

```
provider_payment_id: str

status: str

client_secret: Optional[str]

next_action: Optional[Dict[str, Any]]

charges: List[Dict[str, Any]]

metadata: Dict[str, Any]

created_at: datetime


class PaymentProvider(ABC):

    """Abstract base class for all payment provider implementations."""

    def __init__(self, config: ProviderConfig):
        self.config = config
        self.capabilities = config.capabilities

    @abstractmethod

    async def create_payment_intent(
        self,
        amount: int,
        currency: str,
        metadata: Dict[str, Any],
        idempotency_key: str
    ) -> ProviderPaymentIntent:
        """
        Create payment intent with provider.

        TODO 1: Validate amount and currency against provider limits
        TODO 2: Transform metadata to provider-specific format
        TODO 3: Make API request with idempotency key handling
        TODO 4: Parse response into normalized ProviderPaymentIntent
        """


```

```
TODO 5: Handle provider-specific error codes and retry logic

"""

pass

@abstractmethod

async def create_charge(
    self,
    payment_intent_id: str,
    payment_method_token: str,
    return_url: str
) -> Dict[str, Any]:
    """
    Process charge against payment intent.

    TODO 1: Retrieve payment intent from provider
    TODO 2: Validate payment method compatibility
    TODO 3: Submit charge with authentication handling
    TODO 4: Process 3DS redirect URLs if required
    TODO 5: Return normalized charge response
    """

    pass

@abstractmethod

async def verify_webhook_signature(
    self,
    payload: bytes,
    signature_header: str
) -> bool:
    """
    Verify webhook signature using provider-specific method.

```

```

    TODO 1: Extract signature from header (format varies by provider)

    TODO 2: Calculate expected signature using webhook secret

    TODO 3: Compare signatures using secure comparison method

    TODO 4: Handle provider-specific signature formats (Stripe vs Adyen)

    """
    pass

def supports_capability(self, capability: ProviderCapability) -> bool:
    """Check if provider supports specific capability."""
    return capability in self.capabilities


def calculate_total_cost(self, amount: int, currency: str) -> int:
    """
    Calculate total processing cost including provider fees.

    TODO 1: Look up fee structure for currency

    TODO 2: Calculate percentage-based fees

    TODO 3: Add fixed fees if applicable

    TODO 4: Apply volume discounts if configured

    TODO 5: Return total cost in cents

    """
    pass

```

Provider Selection Engine

The provider selection engine implements the decision logic for routing transactions to optimal providers:

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

from dataclasses import dataclass

from enum import Enum


class SelectionCriteria(Enum):

    LOWEST_COST = "lowest_cost"

    HIGHEST_SUCCESS_RATE = "highest_success_rate"

    FASTEST_PROCESSING = "fastest_processing"

    GEOGRAPHIC_ROUTING = "geographic_routing"

    LOAD_BALANCING = "load_balancing"


@dataclass

class ProviderScore:

    """Scoring breakdown for provider selection decision."""

    provider_name: str

    total_score: float

    cost_score: float

    reliability_score: float

    performance_score: float

    capability_score: float

    disqualification_reason: Optional[str]


class ProviderSelector:

    """Intelligent provider selection based on transaction characteristics."""


    def __init__(self, health_monitor: 'ProviderHealthMonitor'):

        self.health_monitor = health_monitor

        self.selection_weights = {

            'cost': 0.4,

            'reliability': 0.3,

            'performance': 0.2,
```

```
'capability': 0.1

}

async def select_provider(
    self,
    available_providers: List[PaymentProvider],
    amount: int,
    currency: str,
    payment_method: str,
    customer_country: Optional[str] = None
) -> Optional[PaymentProvider]:
    """
    Select optimal provider for transaction.

    TODO 1: Filter providers by hard requirements (currency, amount limits)
    TODO 2: Apply geographic routing rules based on customer location
    TODO 3: Check circuit breaker status for each qualifying provider
    TODO 4: Calculate weighted scores for cost, reliability, performance
    TODO 5: Select highest-scoring available provider
    TODO 6: Log selection decision with full scoring breakdown
    TODO 7: Return selected provider or None if all disqualified
    """
    pass

def _calculate_provider_scores(
    self,
    providers: List[PaymentProvider],
    amount: int,
    currency: str
) -> List[ProviderScore]:
```

```

"""
Calculate detailed scores for provider comparison.

TODO 1: Get current health metrics for each provider
TODO 2: Calculate cost score based on fee structures
TODO 3: Calculate reliability score from success rate history
TODO 4: Calculate performance score from latency metrics
TODO 5: Apply selection weights to create composite scores
TODO 6: Sort providers by total score descending

"""

pass

def _apply_business_rules(
    self,
    providers: List[PaymentProvider],
    transaction_context: Dict[str, Any]
) -> List[PaymentProvider]:
    """
    Apply merchant-specific business rules and compliance filters.

    TODO 1: Check transaction amount against provider limits
    TODO 2: Apply geographic restrictions (EU customers to EU processors)
    TODO 3: Enforce payment method routing rules (Apple Pay to direct integrations)
    TODO 4: Apply compliance requirements (PCI level, regional licensing)
    TODO 5: Return filtered list of qualifying providers

    """

    pass

```

Subscription Billing Implementation

Subscription billing requires careful state management and billing cycle orchestration:

```
from datetime import datetime, timedelta
from typing import Optional, List
from dataclasses import dataclass
from enum import Enum

class SubscriptionStatus(Enum):
    TRIAL = "trial"
    ACTIVE = "active"
    PAST_DUE = "past_due"
    CANCELED = "canceled"
    UNPAID = "unpaid"

class BillingInterval(Enum):
    DAY = "day"
    WEEK = "week"
    MONTH = "month"
    YEAR = "year"

@dataclass
class SubscriptionPlan:
    id: str
    name: str
    amount: int # cents
    currency: str
    interval: BillingInterval
    interval_count: int
    trial_period_days: Optional[int]
    metadata: Dict[str, Any]

class SubscriptionBillingEngine:
    """Core subscription billing orchestration."""
```

```
def __init__(self, payment_processor: 'PaymentProcessor'):

    self.payment_processor = payment_processor


async def create_subscription(
    self,
    customer_id: str,
    plan: SubscriptionPlan,
    payment_method_token: str,
    trial_end: Optional[datetime] = None
) -> 'Subscription':
    """
    Create new subscription with trial handling.

    TODO 1: Validate customer and payment method
    TODO 2: Calculate trial end date if trial period configured
    TODO 3: Create subscription record with TRIAL or ACTIVE status
    TODO 4: Schedule first billing cycle after trial period
    TODO 5: Send subscription confirmation to customer
    TODO 6: Return created subscription with next billing date
    """
    pass


async def process_billing_cycle(
    self,
    subscription: 'Subscription'
) -> 'BillingCycleResult':
    """
    Process subscription billing for current cycle.

    TODO 1: Calculate amount due including usage charges and prorations

```

```
    TODO 2: Create payment intent with subscription metadata

    TODO 3: Attempt payment using stored payment method

    TODO 4: Handle payment failures with retry logic

    TODO 5: Update subscription status based on payment result

    TODO 6: Schedule next billing cycle if payment succeeds

    TODO 7: Send invoice and payment receipts to customer

    """
    pass

def _calculate_proration(
    self,
    subscription: 'Subscription',
    plan_change: Dict[str, Any],
    effective_date: datetime
) -> int:
    """
    Calculate prorated amount for mid-cycle plan changes.

    TODO 1: Calculate unused time on current plan
    TODO 2: Calculate remaining time on new plan
    TODO 3: Compute credit for unused current plan amount
    TODO 4: Compute charge for new plan remainder
    TODO 5: Return net proration amount (positive = charge, negative = credit)
    """
    pass

async def handle_failed_payment(
    self,
    subscription: 'Subscription',
    failure_reason: str
```

```

) -> 'PaymentRetryResult':
    """
    Handle subscription payment failures with smart retry logic.

    TODO 1: Update subscription status to PAST_DUE
    TODO 2: Schedule retry attempts with exponential backoff
    TODO 3: Send payment failure notification to customer
    TODO 4: Update payment method if card expired or invalid
    TODO 5: Cancel subscription after maximum retry attempts
    TODO 6: Process dunning management workflow
    """
    pass

```

Milestone Checkpoints

Each future extension represents a significant development milestone with specific validation checkpoints:

Multi-Provider Integration Checkpoint:

- Command: `python -m pytest tests/providers/ -v`
- Expected: All provider implementations pass interface compliance tests
- Manual verification: Create payment intents through each provider, verify consistent response format
- Success indicators: Provider selection routes transactions correctly, circuit breakers activate during simulated outages

Subscription Billing Checkpoint:

- Command: `python -m pytest tests/subscription/ -v --cov=subscription/`
- Expected: 100% test coverage on billing cycle logic, proration calculations
- Manual verification: Create test subscription, verify trial period handling and automated billing
- Success indicators: Billing cycles process correctly, failed payments trigger appropriate retry sequences

Marketplace Transfer Checkpoint:

- Command: `python -m pytest tests/marketplace/ -v`
- Expected: Transfer orchestration maintains atomicity, fee calculations balance correctly
- Manual verification: Process marketplace payment, verify correct fund distribution to sellers
- Success indicators: All transfers complete successfully, platform fees calculated accurately, audit trail maintained

These extensions transform our basic payment processor into a comprehensive financial platform capable of supporting complex business models while maintaining the reliability and consistency established in our core architecture.

Implementation Guidance

The future extensions outlined above represent significant architectural evolution that requires careful planning, phased implementation, and rigorous testing. Each extension builds upon our existing payment infrastructure while introducing new technical challenges around system integration, data consistency, and operational complexity.

Technology Stack Recommendations

Component	Simple Option	Advanced Option
Provider Abstraction	Factory pattern with direct provider classes	Plugin architecture with dynamic provider loading
Subscription Billing	Cron-based job scheduling with database polling	Event-driven architecture with message queues
Marketplace Transfers	Synchronous transfer processing with database transactions	Asynchronous saga pattern with compensation
Fraud Detection	External API integration with caching	Real-time ML pipeline with feature engineering
Configuration Management	YAML/JSON configuration files	Dynamic configuration service with A/B testing
Monitoring & Alerting	Structured logging with log aggregation	Full observability stack with metrics, traces, logs

Implementation Phases

These extensions should be implemented in carefully planned phases to minimize risk and maintain system stability:

Phase 1: Provider Abstraction Foundation (4-6 weeks)

- Implement base `PaymentProvider` interface and factory pattern
- Create adapter for existing single provider
- Add provider health monitoring infrastructure
- Implement basic provider selection logic
- Add comprehensive testing framework for provider implementations

Phase 2: Multi-Provider Support (6-8 weeks)

- Implement second provider adapter (Stripe + Adyen recommended)
- Build sophisticated provider selection engine with business rules
- Add provider-specific error handling and retry logic
- Implement cross-provider webhook signature verification
- Create provider performance dashboards and alerting

Phase 3: Subscription Billing Core (8-10 weeks)

- Design subscription data model with proper state machine
- Implement subscription creation and lifecycle management

- Build billing cycle processing with automated retries
- Add subscription plan management and change handling
- Create customer communication systems for billing events

Phase 4: Advanced Billing Features (6-8 weeks)

- Implement proration logic for mid-cycle plan changes
- Add usage-based billing and metered subscriptions
- Build dunning management for failed payment recovery
- Add subscription analytics and churn prediction
- Implement subscription pause/resume functionality

Phase 5: Marketplace Infrastructure (10-12 weeks)

- Design connected account and KYC verification workflows
- Implement multi-party payment processing with transfer groups
- Build platform fee calculation and collection systems
- Add marketplace-specific compliance and reporting
- Create seller dashboard and payout management

Phase 6: Fraud Detection Integration (8-10 weeks)

- Implement real-time risk scoring pipeline
- Add device fingerprinting and behavioral analysis
- Build rule-based fraud detection with machine learning enhancement
- Create fraud analyst tools and case management
- Add automated response systems for high-risk transactions

Core Infrastructure Requirements

These extensions require significant infrastructure enhancements to support increased complexity and scale:

```
# Enhanced configuration management for multi-provider support

@dataclass

class SystemConfig:

    """Centralized configuration for extended payment system."""

    # Provider configurations

    providers: Dict[str, ProviderConfig]

    default_provider: str

    provider_selection_strategy: SelectionStrategy

    # Subscription billing settings

    billing_retry_attempts: int = 3

    billing_retry_intervals: List[int] = [1, 3, 7] # days

    trial_grace_period: int = 1 # days

    # Marketplace settings

    platform_fee_percentage: float = 2.9

    payout_schedule: str = "weekly" # daily, weekly, monthly

    kyc_verification_required: bool = True

    # Fraud detection thresholds

    risk_score_decline_threshold: float = 0.8

    risk_score_review_threshold: float = 0.6

    velocity_limits: Dict[str, int] = None

    # System reliability settings

    circuit_breaker_config: CircuitBreakerConfig = None

    retry_config: RetryConfig = None

    reconciliation_schedule: str = "0 2 * * *" # daily at 2 AM
```

```
# Enhanced monitoring for complex system operations

class SystemMonitor:

    """Comprehensive monitoring for extended payment system."""

    async def track_provider_performance(
        self,
        provider_name: str,
        operation: str,
        duration: float,
        success: bool,
        error_code: Optional[str] = None
    ):
        """
        Track provider operation metrics for health monitoring.

        TODO 1: Record operation timing and success metrics
        TODO 2: Update provider health scores in real-time
        TODO 3: Trigger alerts if error rates exceed thresholds
        TODO 4: Log detailed operation context for debugging
        """

        pass

    async def track_subscription_metrics(
        self,
        event_type: str, # created, billed, failed, canceled
        subscription_id: str,
        plan_id: str,
        amount: int,
        metadata: Dict[str, Any]
    ):
```

```
"""
Track subscription business metrics for analysis.

TODO 1: Record subscription lifecycle events
TODO 2: Calculate MRR (Monthly Recurring Revenue) impact
TODO 3: Track churn rate and customer lifetime value
TODO 4: Update subscription health dashboards
"""

pass
```

```
async def track_fraud_decision(
    self,
    payment_intent_id: str,
    risk_score: float,
    decision: str, # approved, declined, review
    risk_factors: List[str],
    processing_time: float
):
    """
    Track fraud detection decisions for model improvement.

    TODO 1: Log fraud decision with full risk analysis context
    TODO 2: Track false positive and false negative rates
    TODO 3: Feed decision outcomes back to ML models
    TODO 4: Alert on unusual fraud pattern detection
    """

    pass
```

```
# Background job processing for subscription billing
class BillingJobProcessor:
```

```
"""Background processing for subscription billing operations."""

def __init__(self, subscription_engine: SubscriptionBillingEngine):
    self.subscription_engine = subscription_engine

async def process_daily_billing_cycles(self):
    """
    Process all subscriptions due for billing today.

    TODO 1: Query subscriptions with billing date = today
    TODO 2: Process each subscription with error handling
    TODO 3: Track billing success/failure rates
    TODO 4: Schedule retry jobs for failed billing attempts
    TODO 5: Update billing cycle completion metrics
    """
    pass

async def process_trial_expirations(self):
    """
    Handle trial period expirations and first billing.

    TODO 1: Query subscriptions with trial ending today
    TODO 2: Transition subscription status from trial to active
    TODO 3: Process first billing cycle with trial-to-paid conversion
    TODO 4: Send trial expiration notifications to customers
    TODO 5: Handle payment failures for trial conversions
    """
    pass

async def process_subscription_retries(self):
```

```
....
```

```
Retry failed subscription billing attempts.
```

```
TODO 1: Query subscriptions in past_due status eligible for retry
```

```
TODO 2: Apply exponential backoff retry schedule
```

```
TODO 3: Attempt billing with updated payment methods if available
```

```
TODO 4: Cancel subscriptions exceeding maximum retry attempts
```

```
TODO 5: Send appropriate notifications based on retry outcomes
```

```
....
```

```
pass
```

Testing Strategy for Extensions

Each extension requires comprehensive testing covering integration, performance, and business logic validation:

```
# Multi-provider testing infrastructure

class TestProviderAbstraction:

    """Test suite for provider abstraction layer."""

    async def test_provider_interface_compliance(self):

        """Verify all providers implement required interface methods."""

        # TODO: Instantiate each provider implementation

        # TODO: Verify all interface methods are implemented

        # TODO: Test method signatures match interface definition

        # TODO: Validate provider capabilities declaration accuracy

        pass

    async def test_provider_selection_logic(self):

        """Test provider selection under various scenarios."""

        # TODO: Mock provider health metrics and capabilities

        # TODO: Test selection with different transaction characteristics

        # TODO: Verify business rule application (geographic, amount limits)

        # TODO: Test fallback behavior when primary provider unavailable

        pass

    async def test_cross_provider_consistency(self):

        """Verify consistent behavior across provider implementations."""

        # TODO: Process identical transactions through each provider

        # TODO: Verify webhook event format normalization

        # TODO: Test error handling consistency across providers

        # TODO: Validate idempotency behavior with each provider

        pass

    # Subscription billing test scenarios

class TestSubscriptionBilling:
```

```
"""Comprehensive test suite for subscription billing logic."""

async def test_billing_cycle_processing(self):
    """Test complete billing cycle from charge to next cycle scheduling."""

    # TODO: Create subscription with test payment method

    # TODO: Mock billing cycle due date

    # TODO: Process billing and verify payment intent creation

    # TODO: Confirm next cycle scheduling accuracy

    pass

async def test_proration_calculations(self):
    """Verify proration accuracy for plan changes."""

    # TODO: Create subscription with monthly plan

    # TODO: Upgrade plan mid-cycle with known timing

    # TODO: Calculate expected proration amount manually

    # TODO: Compare system calculation with expected result

    pass

async def test_failed_payment_handling(self):
    """Test subscription failure and retry workflows."""

    # TODO: Create subscription with failing payment method

    # TODO: Trigger billing cycle and capture failure

    # TODO: Verify retry scheduling and status updates

    # TODO: Test subscription cancellation after max retries

    pass

# Integration testing for complete workflows

class TestEndToEndWorkflows:

    """Integration tests for complete payment workflows."""

```

```
async def test_marketplace_payment_distribution(self):

    """Test complete marketplace payment with multi-party transfers."""

    # TODO: Create marketplace transaction with multiple sellers

    # TODO: Process payment through selected provider

    # TODO: Verify transfer distribution and platform fee collection

    # TODO: Confirm all parties receive correct amounts

    pass
```

```
async def test_subscription_with_fraud_detection(self):

    """Test subscription creation with fraud screening."""

    # TODO: Create high-risk subscription attempt

    # TODO: Verify fraud detection triggers appropriate response

    # TODO: Test subscription creation after fraud clearance

    # TODO: Confirm billing cycle processing with fraud monitoring

    pass
```

Production Deployment Considerations

Deploying these extensions in production requires careful attention to operational concerns:

Database Migration Strategy:

- Plan schema changes to support new entities (Subscription, ConnectedAccount, Transfer)
- Implement backward-compatible migrations allowing gradual rollout
- Add proper indexing for subscription billing queries and transfer lookups
- Consider database partitioning for high-volume subscription and transaction data

Performance Monitoring:

- Add metrics tracking for provider selection decision time
- Monitor subscription billing job processing throughput
- Track fraud detection analysis latency impact on transaction flow
- Implement alerting for extension-specific failure scenarios

Operational Runbooks:

- Document provider switching procedures for outages or performance issues
- Create subscription billing troubleshooting guides for failed payments
- Establish fraud detection tuning procedures for false positive optimization
- Define marketplace payout reconciliation and dispute resolution processes

These future extensions represent the evolution from a basic payment processor to a comprehensive financial platform capable of supporting sophisticated business models while maintaining the reliability and consistency established in our foundational architecture.

Glossary

Milestone(s): All milestones - understanding payment industry terminology is essential for payment intent & idempotency (Milestone 1), payment processing & 3DS (Milestone 2), refunds & disputes (Milestone 3), and webhook reconciliation (Milestone 4)

Think of this glossary as a payment industry dictionary and technical reference. Just as financial professionals use precise terminology to avoid costly misunderstandings, payment systems require exact vocabulary to ensure clear communication between developers, payment processors, banks, and regulatory bodies. Every term has specific legal, technical, or business implications that affect how the system behaves.

This glossary serves multiple audiences: developers implementing payment functionality, QA engineers writing test scenarios, product managers defining requirements, and support teams diagnosing issues. Each definition includes both the conceptual meaning and the practical implementation context within our payment gateway system.

Payment Industry Core Concepts

Payment intent represents a customer's commitment to pay before any money actually moves. Think of it like placing an order at a restaurant - you've committed to pay for the meal, but the actual charge happens later when you settle the bill. Payment intents allow merchants to capture customer commitment early, handle authentication flows, and ensure the payment amount is reserved while completing additional verification steps.

In our system, payment intents serve as the central coordination point for all payment-related activities. They maintain state throughout the entire payment lifecycle, from initial creation through final settlement or cancellation. The payment intent tracks not just the monetary amount, but also metadata, authentication requirements, and relationship to downstream charges and refunds.

Idempotency key functions as a unique fingerprint for payment operations, ensuring that duplicate requests produce identical results without unwanted side effects. Consider the analogy of a check number - banks use check numbers to detect and prevent duplicate processing of the same check. Similarly, idempotency keys prevent scenarios where network retries or user interface double-clicks result in charging customers multiple times for the same purchase.

Our idempotency implementation goes beyond simple duplicate detection. The system stores a cryptographic hash of the request parameters along with the idempotency key, ensuring that reusing an idempotency key with different payment parameters correctly triggers an error rather than returning the cached response from the original request.

3D Secure (Three Domain Secure) implements Strong Customer Authentication protocol required by European PSD2 regulations and increasingly adopted globally. The three domains are the issuer domain (customer's bank), acquirer domain (merchant's bank), and interoperability domain (payment networks like Visa and Mastercard that facilitate communication between the other domains).

The 3D Secure flow resembles a secure handshake between the customer's bank and the merchant. When a payment requires authentication, the customer is redirected to their bank's secure authentication page, completes verification

(password, SMS code, biometric), and returns to the merchant with a cryptographic proof of authentication. Our system handles this redirect flow while maintaining payment state and security throughout the authentication journey.

Tokenization replaces sensitive payment data with mathematically unrelated tokens that have no intrinsic value. Imagine a coat check system at a restaurant - you receive a numbered ticket (token) that represents your coat, but the ticket itself isn't your coat and is worthless to anyone who doesn't have access to the coat check system.

Payment tokenization works similarly. When customers enter credit card details, the payment processor immediately replaces the card number with a random token. Our system stores and operates on tokens exclusively, never handling actual card numbers. This dramatically reduces PCI compliance scope since we never touch sensitive cardholder data.

Webhook reconciliation ensures that our local payment records remain synchronized with the authoritative state maintained by external payment providers. Think of it like balancing a checkbook - periodically comparing your records with the bank's records to identify and resolve any discrepancies.

Payment providers send webhook notifications when payment states change, but webhooks can fail, arrive out of order, or be delayed. Our reconciliation process acts as a safety net, regularly comparing our local payment states with the provider's authoritative records and resolving any inconsistencies through automated correction or manual review workflows.

Payment Processing Terminology

Terminal state describes payment statuses that allow no further transitions - the payment has reached a final, unchangeable outcome. Like a door that can be opened or closed but not "half-open," payments eventually reach states like `succeeded`, `failed`, or `canceled` from which no further state changes are possible.

Understanding terminal states is crucial for payment system design because it determines when resources can be cleaned up, when customers can be charged for goods, and when refunds become the only mechanism for returning money. Our state machine design explicitly prevents illegal transitions from terminal states to maintain data integrity.

Chargeback represents a forced payment reversal initiated by the customer's bank, typically when the customer disputes a transaction. Unlike refunds (which merchants control), chargebacks are imposed by the payment network and cannot be prevented - only contested through a formal dispute process.

Chargebacks carry additional fees and penalties beyond the reversed transaction amount. When a chargeback occurs, the merchant loses both the original payment amount and pays a chargeback fee (typically \$15-25). Excessive chargeback rates can result in higher processing fees or account termination by payment processors.

PCI DSS (Payment Card Industry Data Security Standard) establishes security requirements for any organization that handles, stores, or transmits credit card information. Think of PCI DSS as building codes for payment systems - mandatory safety requirements that prevent catastrophic failures (data breaches) that could harm customers and destroy businesses.

Our system achieves PCI compliance through tokenization, which eliminates the need to store sensitive cardholder data. By delegating card data handling to PCI-compliant payment processors and working exclusively with tokens, we significantly reduce our compliance scope and security risk.

Technical System Concepts

Request parameter hash creates a deterministic fingerprint of essential payment operation parameters, enabling detection of duplicate requests with different parameter values. Like a document checksum that changes if even one

character is modified, the request parameter hash ensures that reusing an idempotency key with different payment amounts, currencies, or payment methods correctly triggers an error.

The hash calculation includes all parameters that affect payment outcomes: amount, currency, payment method token, metadata, and return URLs. This prevents malicious or accidental attempts to reuse idempotency keys for different transactions while allowing legitimate retry scenarios to proceed safely.

Circuit breaker implements failure protection that prevents cascading failures by temporarily blocking calls to failing external services. Imagine a electrical circuit breaker in your home - when it detects dangerous conditions (electrical overload), it immediately cuts power to prevent fires, then can be reset once the dangerous condition is resolved.

Payment circuit breakers monitor error rates and response times for external provider APIs. When failure thresholds are exceeded, the circuit breaker transitions to an "open" state, immediately failing requests without attempting provider calls. After a timeout period, it enters a "half-open" state to test if the provider has recovered.

Exponential backoff implements a retry delay strategy where the wait time between retry attempts increases exponentially (1 second, 2 seconds, 4 seconds, 8 seconds, etc.). This prevents overwhelming failing services with retry attempts while giving them time to recover.

Our implementation includes jitter - random variance added to retry delays to prevent the "thundering herd" problem where many clients retry simultaneously. Instead of all clients retrying exactly every 4 seconds, they retry at random intervals between 3-5 seconds, distributing the load more evenly.

Compensation transaction executes the inverse operation needed to maintain system consistency after partial failures in distributed operations. Like the "undo" function in a word processor, compensation transactions reverse the effects of operations that cannot be completed successfully.

For example, if a payment succeeds with the provider but our webhook processing fails, a compensation transaction might involve calling the provider to refund the payment or marking our local records for manual reconciliation. These transactions ensure that partial failures don't leave the system in inconsistent states.

State Management and Consistency

Eventual consistency describes a consistency model where distributed systems converge to a consistent state over time, even though they may be temporarily inconsistent during normal operation. Think of it like a group text conversation where messages arrive in slightly different orders on different phones, but eventually everyone sees the same conversation history.

Payment systems naturally exhibit eventual consistency because they span multiple external services (payment providers, banks, card networks) that process transactions at different speeds. Our system design embraces this reality by implementing reconciliation processes that detect and resolve temporary inconsistencies automatically.

Reconciliation encompasses the process of synchronizing local payment state with external provider state to detect and resolve discrepancies. Like auditors comparing company books with bank statements, reconciliation identifies differences between what we think happened and what actually happened according to authoritative sources.

Our reconciliation system operates continuously through webhook processing and periodically through batch comparison jobs. It maintains detailed logs of discrepancies, resolution actions taken, and manual review requirements for complex cases that cannot be automatically resolved.

Jitter adds random variance to timing operations, particularly retry delays and scheduled job execution, to prevent synchronized behavior that can overwhelm systems. Without jitter, all clients experiencing the same failure might retry at

exactly the same intervals, creating traffic spikes that make recovery more difficult.

Our jitter implementation uses cryptographically secure random number generation to ensure unpredictable variance. For retry delays, we typically add $\pm 25\%$ jitter, so a 4-second delay becomes a random interval between 3-5 seconds.

Advanced Payment Concepts

Provider abstraction layer creates a translation boundary between our domain model and provider-specific implementations, enabling support for multiple payment processors through unified interfaces. Think of it like a universal translator that allows our system to communicate with different payment providers despite their varying APIs and data formats.

The abstraction layer standardizes concepts like payment intents, charges, and refunds across providers while handling provider-specific peculiarities like different authentication methods, webhook formats, and error codes. This design enables switching providers or supporting multiple providers simultaneously without changing core business logic.

Provider selection strategy implements the decision engine for routing transactions to optimal providers based on various criteria like cost, success rates, geographic coverage, and capabilities. Similar to GPS routing algorithms that consider traffic, road conditions, and distance to find optimal paths, provider selection algorithms evaluate multiple factors to maximize transaction success while minimizing costs.

Our selection strategy considers transaction characteristics (amount, currency, payment method), customer attributes (geographic location, risk profile), and real-time provider performance metrics to make intelligent routing decisions.

Connected accounts represent verified seller accounts in marketplace scenarios where the platform facilitates payments between buyers and sellers. Like verified merchant accounts at a shopping mall where the mall management has vetted each store, connected accounts undergo verification and capability checks before being authorized to receive payments.

Connected accounts maintain their own verification status, supported capabilities, and compliance requirements. The platform can enable or disable specific capabilities (card payments, bank transfers, instant payouts) based on the account's verification level and risk assessment.

Transfer groups organize related money movements from a single marketplace payment into multiple destination accounts, typically including seller payments and platform fees. Think of it like a payroll system that takes one large payment from a company and distributes it to multiple employees while deducting taxes and benefits.

Transfer groups maintain atomicity - either all transfers in the group succeed, or none do. This prevents scenarios where seller payments succeed but platform fee collection fails, leaving the marketplace unable to collect revenue from completed transactions.

Subscription and Recurring Billing

Subscription billing engine orchestrates the complex workflows required for recurring payment processing, including trial periods, plan changes, proration calculations, failed payment recovery, and lifecycle management. Like a sophisticated scheduling system that manages recurring appointments while handling cancellations, rescheduling, and special requirements.

The billing engine coordinates multiple subsystems: customer management, payment method storage, invoice generation, payment processing, dunning management (handling failed payments), and customer communication. It ensures that subscription charges occur at the correct times with appropriate amounts while gracefully handling edge cases.

Billing cycle represents an individual billing period with associated charges, payment attempts, and status tracking. Each cycle maintains its own payment state independent of other cycles, allowing for complex scenarios like partial payments, billing adjustments, and cycle-specific promotions.

Billing cycles handle proration automatically when customers upgrade, downgrade, or change billing frequency mid-cycle. The system calculates precise amounts based on usage periods and applies appropriate credits or additional charges.

Proration calculates proportional billing adjustments when subscription changes occur mid-cycle, ensuring customers pay exactly for the service period they actually use. Like calculating utility bills that account for mid-month service changes, proration ensures fair charging regardless of when changes occur.

Our proration engine handles complex scenarios including plan upgrades, downgrades, quantity changes, and billing frequency modifications. It generates detailed line items showing original charges, credits for unused time, and additional charges for new services.

Risk Management and Fraud Detection

Risk scoring pipeline analyzes transaction patterns, customer behavior, and contextual factors to identify potentially fraudulent payments before they complete processing. Similar to credit scoring systems that evaluate loan applications, risk scoring examines multiple data points to assign fraud probability scores.

The pipeline operates in real-time during payment processing, evaluating factors like transaction velocity, geographic inconsistencies, device fingerprinting, and behavioral patterns. High-risk transactions can be automatically declined, flagged for manual review, or subjected to additional authentication requirements.

Provider health monitoring continuously tracks provider performance metrics including success rates, response times, error patterns, and service availability to inform routing decisions and detect degraded service conditions. Like network monitoring systems that track server health, provider monitoring ensures optimal transaction routing and early problem detection.

Health monitoring aggregates metrics across different time windows (real-time, hourly, daily) and transaction types (small vs large amounts, different currencies, various payment methods) to provide comprehensive visibility into provider performance characteristics.

Compliance and Security Standards

SAQ A (Self-Assessment Questionnaire A) represents the simplest PCI DSS compliance requirement for merchants who have completely outsourced payment processing and never handle cardholder data directly. Think of it like a basic safety checklist for businesses that don't directly handle dangerous materials but still need to maintain safe practices.

Merchants qualifying for SAQ A only need to complete a short questionnaire (about 12 questions) rather than undergoing comprehensive security audits. Our tokenization-based approach enables merchants to qualify for SAQ A by ensuring they never touch sensitive cardholder data.

SAQ A-EP (SAQ A with E-commerce Payment processing) applies to merchants who use hosted payment forms but maintain some payment-related functions on their systems. This requires additional security controls around network segmentation, access controls, and system monitoring while still avoiding direct cardholder data handling.

SAQ D represents the most comprehensive PCI assessment required for merchants who store, process, or transmit cardholder data directly. This involves extensive security audits, penetration testing, and ongoing compliance monitoring - the equivalent of full regulatory oversight for high-risk activities.

TLS 1.2 specifies the minimum required Transport Layer Security version for protecting payment data in transit. Like requiring reinforced security trucks for valuable cargo transport, TLS 1.2 ensures that payment information cannot be intercepted or modified during network transmission.

Our system enforces TLS 1.2 for all payment-related communications and rejects connections using older, vulnerable protocols. This includes connections to payment providers, webhook endpoints, and customer-facing payment forms.

Implementation Guidance

The payment industry terminology forms the foundation for clear communication throughout system implementation. Understanding these concepts enables precise technical discussions, accurate requirement specifications, and effective debugging when issues arise.

Payment Domain Language Usage

When implementing payment systems, consistent terminology usage prevents misunderstandings that could lead to incorrect business logic or security vulnerabilities. Use these preferred terms consistently:

Concept	Preferred Term	Avoid	Reason
Customer commitment to pay	payment intent	payment request, order	Aligns with industry standards
Unique operation identifier	idempotency key	transaction id, request id	Specifies duplicate prevention purpose
Card authentication protocol	3D Secure	3DS, three-d-secure	Official protocol name
State synchronization	webhook reconciliation	webhook processing	Emphasizes consistency checking
Sensitive data replacement	tokenization	encryption, hashing	Describes specific security technique
Forced payment reversal	chargeback	dispute, reversal	Distinguishes from merchant-initiated refunds
Security compliance framework	PCI DSS	PCI compliance	Complete standard name
Final payment status	terminal state	end state, final status	Payment industry convention

Technical Term Application

Understanding when and how to apply technical terms correctly prevents implementation errors:

Idempotency applies to any operation that can be safely retried without changing the result. In payment systems, this includes payment intent creation, charge processing, refund requests, and webhook event processing. However, not all operations are naturally idempotent - some require explicit design to achieve idempotent behavior.

Eventual consistency describes the expected behavior for distributed payment systems. Developers should design for temporary inconsistencies and implement reconciliation processes rather than expecting immediate consistency across all system components.

Circuit breakers should be applied to all external service calls, including payment provider APIs, database connections, and webhook deliveries. However, circuit breakers must be configured appropriately for each service's characteristics -

payment processing might tolerate longer timeout periods than user-facing API calls.

Error Message and Log Terminology

Use precise terminology in error messages, log entries, and monitoring alerts to facilitate rapid debugging:

```
Good: "Payment intent pi_abc123 failed 3D Secure authentication - customer abandoned auth flow"  
Poor: "Payment failed - authentication error"
```

```
Good: "Webhook reconciliation detected state mismatch: local=processing, provider=succeeded"  
Poor: "Data sync error"
```

```
Good: "Circuit breaker opened for provider stripe-api after 5 consecutive timeout failures"  
Poor: "External service unavailable"
```

Domain Model Consistency

Maintain terminology consistency throughout the codebase by establishing clear mappings between domain concepts and implementation artifacts:

Domain Concept	Code Representation	Database Schema	API Response
Payment intent	<code>PaymentIntent</code> class	<code>payment_intents</code> table	<code>payment_intent</code> object
Idempotency key	<code>idempotency_key</code> field	<code>idempotency_keys</code> table	<code>idempotency_key</code> property
3D Secure flow	<code>handle_three_ds_return</code> method	<code>three_ds_authentication</code> table	<code>next_action.redirect_to_url</code>
Webhook reconciliation	<code>reconcile_payment_state</code> function	<code>reconciliation_runs</code> table	Not exposed externally

This consistency enables new team members to understand the system quickly and reduces cognitive overhead when switching between different system layers.

Compliance Communication

When communicating with compliance teams, auditors, or legal counsel, use official terminology from relevant standards and regulations:

- Reference "PCI DSS requirements" not "PCI rules"
- Specify "Strong Customer Authentication (SCA)" not "European payment authentication"
- Use "cardholder data environment (CDE)" not "payment processing system"
- Distinguish "stored cardholder data" from "payment transaction data"

Integration Documentation

When documenting integrations with external payment providers, maintain clear separation between our domain terminology and provider-specific terminology through translation tables:

Our Term	Stripe Term	PayPal Term	Square Term
Payment intent	Payment Intent	Order	Payment
Charge	Charge	Capture	Payment
Refund	Refund	Refund	Refund
Webhook event	Event	Webhook	Webhook

This approach enables supporting multiple providers while maintaining consistent internal terminology throughout our system implementation.