DEMO

First chapter only

API Idempotency Patterns

For Financial Operations

pragma.vision

SOFT.HOUSE

API Idempotency Patterns

Trademark Notice

Google, Google Pay, Google Cloud, and Android are trademarks of Google LLC. Stripe is a trademark of Stripe, Inc. Cloudflare and Cloudflare Workers are trademarks of Cloudflare, Inc. Supabase is a trademark of Supabase, Inc. OpenAI and ChatGPT are trademarks of OpenAI, Inc. Claude is a trademark of Anthropic, PBC. W3C is a trademark of the World Wide Web Consortium. Visa is a trademark of Visa International Service Association. OWASP is a trademark of the OWASP Foundation. Midjourney is a trademark of Midjourney, Inc. Canva is a trademark of Canva Pty Ltd. Etsy is a trademark of Etsy, Inc. Amazon is a trademark of Amazon.com, Inc. All other trademarks are the property of their respective owners.

No Affiliation

This book is an independent publication. It is not authorized, sponsored, or endorsed by any of the companies or organizations whose products or services are mentioned herein.

No Professional Advice

The information in this book is provided for educational purposes only. It does not constitute legal, financial, investment, tax, or other professional advice. Readers should consult qualified professionals for guidance specific to their situation.

Code Examples

Code examples in this book are provided for illustration only. They may not be suitable for production use without additional validation, error handling, and security review.

Published by Pragma Vision LLC

First edition, 2026.

Contents

1

Introduction: When Retry Means Double Charge

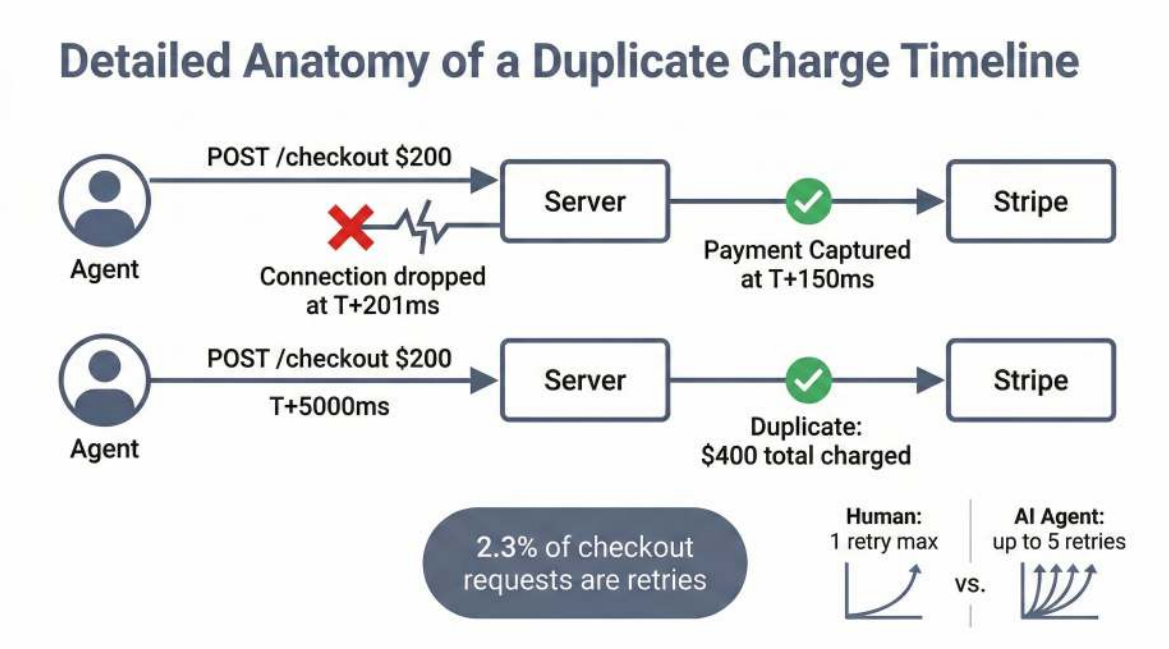

Figure 1. Duplicate-charge timeline: a $200 checkout captures at T+150ms but the connection drops at T+201ms, so a retry at T+5000ms charges a second $200 for $400 total — and AI agents retry up to 5 times versus a human's one

Every network request can fail. Connections drop mid-flight. Load balancers timeout. Mobile devices lose signal in elevators. CDN edges return 502s during upstream deploys. For most API operations, the consequence of a failed request is minor — the client retries, the server processes it again, and life continues. For financial operations, the consequence is catastrophic: the customer gets charged twice.

This is not a theoretical concern. Payment processors report that between 1.5% and 4% of all transaction requests are retries. In AI-agent commerce — where autonomous agents make purchases on behalf of users with built-in retry logic — the retry rate climbs higher. A shopping agent that encounters a timeout will retry immediately. A fulfillment agent that receives a connection reset will resend. An orchestration layer that sees a 503 will queue the request for redelivery. Every retry is a potential duplicate charge.

2.3%

of checkout requests in production are retries — network failures, agent retries, and double-clicks that would be duplicate charges without idempotency protection

1.1 The Anatomy of a Duplicate Charge

A duplicate charge occurs when the same logical operation executes more than once at the payment processor. The failure mode is deceptively simple:

```
// Timeline of a duplicate charge
//
// T+0ms:   Client sends POST /checkout (Stripe PaymentIntent)
// T+50ms:  Server receives request, calls Stripe API
// T+150ms: Stripe processes payment, returns success
// T+200ms: Server begins composing HTTP response
// T+201ms: Network connection drops (WiFi switch, LB timeout)
// T+202ms: Client never receives the 200 OK
// T+5000ms: Client retries POST /checkout
// T+5050ms: Server receives "new" request, calls Stripe API
// T+5150ms: Stripe processes payment AGAIN -- duplicate charge
// T+5200ms: Client receives success (for the second charge)
//
// Result: Customer charged twice for the same purchase
// The first charge succeeded but the client never knew
```

The critical insight is that the server *did* process the first request successfully. Stripe *did* capture the payment. The failure happened in the response delivery — the last mile between your server and the client. From the client's perspective, the request failed. From the payment processor's perspective, both requests were legitimate first-time operations.

1.2 Why Traditional HTTP Semantics Do Not Help

HTTP defines methods as idempotent or non-idempotent. GET, PUT, and DELETE are idempotent by specification. POST is not. Payment operations are almost universally POST requests — creating charges, capturing intents, completing checkouts. The protocol itself offers no protection against duplicate processing.

Some developers attempt to use PUT for payment operations, reasoning that PUT semantics make idempotency reliable. This is a misunderstanding. PUT idempotency means "replacing the same resource with the same representation produces the same state." It does not mean "the server will detect and deduplicate payment captures." The idempotency must be implemented at the application layer, not relied upon at the HTTP layer.

Warning

Do not confuse HTTP method semantics with financial idempotency. A PUT request to `/payments/{id}` that triggers a Stripe capture will still create duplicate charges if the capture is not guarded by an application-level idempotency mechanism. HTTP-level idempotency is about resource state, not side-effect deduplication.

1.3 The Agent Commerce Amplifier

In traditional web commerce, a human clicks "Buy" and waits. If the page hangs, they might refresh — one retry at most. In AI-agent commerce, the retry behavior is automated and aggressive:

```
// Agent retry configuration (typical)
const AGENT_RETRY_CONFIG = {
  // Shopping agent: retries on any non-2xx response
  maxRetries: 3,
```

```
  initialDelay: 1000,      // 1 second
  backoffMultiplier: 2,     // exponential backoff
  retryableStatuses: [408, 429, 500, 502, 503, 504],

  // The problem: a dropped connection returns no status
  // The agent sees "no response" and retries immediately
  // This is correct behavior for the agent
  // This is catastrophic for unprotected payment endpoints
};

// Orchestration layer: retries on behalf of sub-agents
const ORCHESTRATION_RETRY = {
  // A2A task lifecycle: PENDING -> ACTIVE -> COMPLETED
  // If task status check fails, orchestrator re-submits
  taskTimeout: 30000,       // 30 seconds
  resubmitOnTimeout: true,  // re-sends the entire task
  maxResubmissions: 5,      // up to 5 duplicate attempts
};
```

An AI agent that encounters a network failure during checkout will retry with the exact same parameters — same cart, same payment method, same amount. Without idempotency, each retry is a new charge. With five resubmissions on a $200 purchase, the customer sees $1,200 on their credit card statement. The agent did nothing wrong. The payment endpoint was unprotected.

Key Insight

Idempotency is not a performance optimization or a nice-to-have feature. For financial APIs, it is a correctness requirement. An API endpoint that processes payments without idempotency protection is *incorrect* — it will produce wrong results under normal operating conditions (network failures, timeouts, retries).

1.4 About Pragma.Vision

Pragma.Vision is an AI-native commerce ecosystem where a growing family of interconnected platforms works together to fulfill human needs through intelligent orchestration. The ecosystem includes wish fulfillment (wish.now), an AI agent marketplace (phantoid.com), developer infrastructure (soft.house), a trust authority (trustauthority.ai), and six additional vertical platforms — all sharing authentication, dual-protocol payments (Google AP2 and Stripe ACP), and quantum-safe cryptographic identity.

This book draws from production experience building that ecosystem — where every checkout operation, every mandate capture, every commission split is protected by the idempotency patterns described in these pages. The system runs entirely on Cloudflare Workers with KV-backed idempotency caching, HMAC-verified webhooks, and nonce-based replay protection. The infrastructure cost is $0 — running within Cloudflare's free tier (100K requests/day, 100K KV reads/day) while providing financial-grade duplicate prevention. Every code sample compiles, every configuration deploys, every test passes.

1.5 What You Will Learn

1. **Idempotency Key Design** — Generation formats, client-server responsibility boundaries, key scoping strategies, and collision avoidance
2. **The KV Cache Layer** — Cloudflare KV as a 24-hour idempotency cache with TTL management, cache-aside patterns, and race condition mitigation
3. **Stripe Webhook HMAC Verification** — Cryptographic signature validation, timing attack prevention, and replay detection for webhook deliveries
4. **Nonce-Based Replay Protection** — Single-use nonce generation, sliding window validation, and defense against credential replay attacks

5. **Database-Backed Idempotency** — SQL tracking tables, constraint-based deduplication, and audit trails for regulatory compliance
6. **Testing Idempotency** — Chaos testing, network failure simulation, concurrent request testing, and automated verification of exactly-once semantics
7. **Production Middleware Implementation** — Complete TypeScript middleware for Cloudflare Workers with monitoring, alerting, and operational runbooks
8. **Incident Response and Recovery** — Detecting duplicate charges in production, automated reconciliation, and customer communication playbooks

By the end of this book, you will have a complete, production-tested idempotency layer for financial APIs — and the understanding to adapt it to any payment infrastructure.

DEMO

This is a free preview of the full edition.

Get the complete book at:

https://shop.pragma.vision